5. 합성곱 신경망 이해

- 합성곱 연산에 대해 이해한다.
- 합성곱 신경망을 이해하여 구현한다.
- 케라스로 합성곱 신경망을 구현한다.

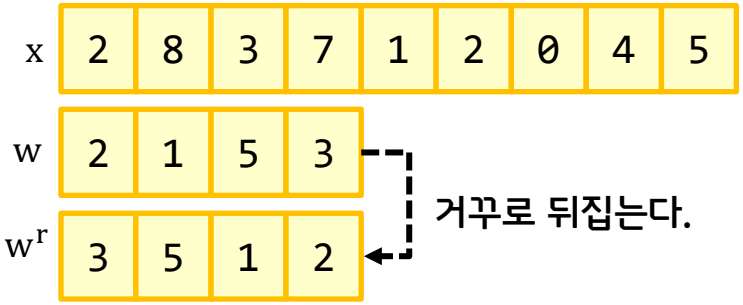
5. 합성곱 신경망 이해

5.1 합성곱 연산

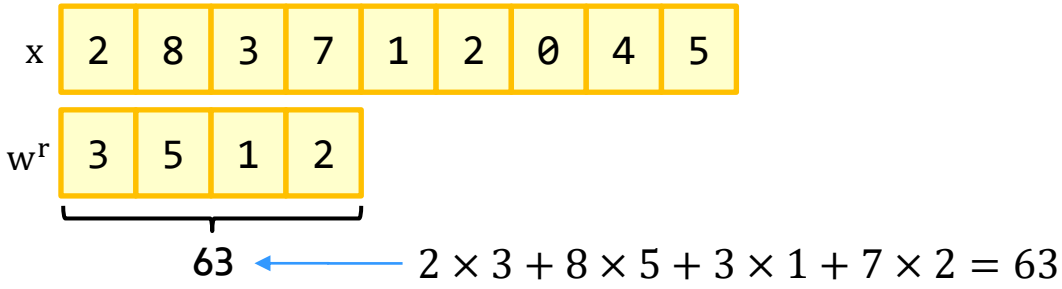
5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 구현

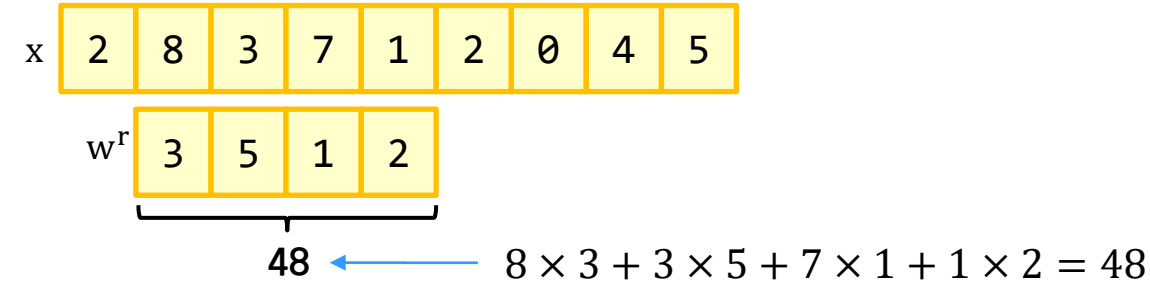
배열 하나 선택해 뒤집기



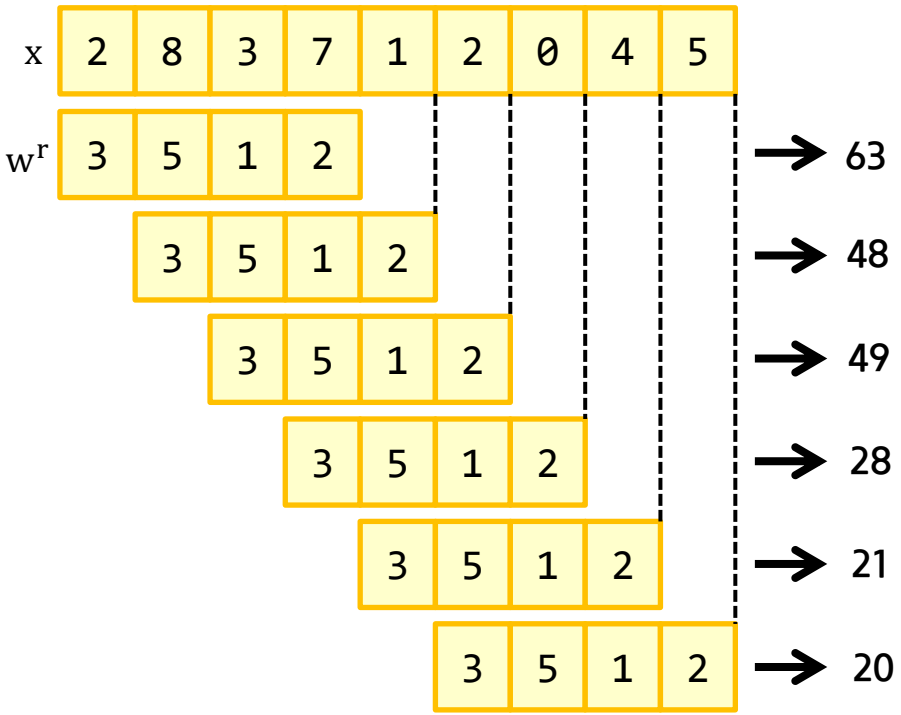
첫 번째 합성곱



두 번째 합성곱



전체 합성곱



합성곱 구현

```
import numpy as np
x = np.array([2, 8, 3, 7, 1, 2, 0, 4, 5])
w = np.array([2, 1, 5, 3])
```

flip() 함수를 이용한 배열 뒤집기

```
w_r = np.flip(w)
print(w_r)
```

[3 5 1 2]

넘파이의 점 곱으로 합성곱 연산

```
for i in range(6):
    print(np.dot(x[i:i+4], w_r.reshape(-1,1)))
```

[63]
[48]
[49]
[28]
[21]
[20]

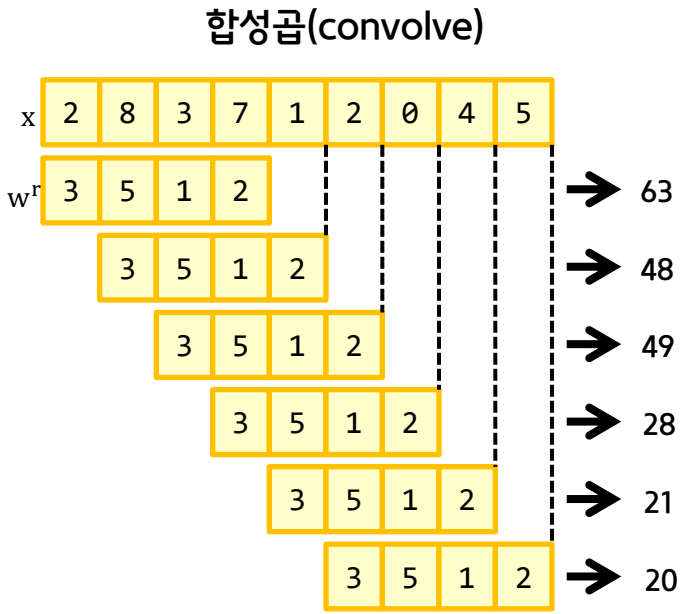
$$\begin{bmatrix} 2 & 8 & 3 & 7 \\ 8 & 3 & 7 & 1 \\ 3 & 7 & 1 & 2 \\ 7 & 1 & 2 & 0 \\ 1 & 2 & 0 & 4 \\ 2 & 0 & 4 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 5 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 63 \\ 48 \\ 49 \\ 28 \\ 21 \\ 20 \end{bmatrix}$$

싸이파이로 합성곱 수행

```
from scipy.signal import convolve
convolve(x, w, mode='valid')
```

array([63, 48, 49, 28, 21, 20])

합성곱 신경망은 진짜 합성곱을 사용하지 않는다.
합성곱 대신 교차상관을 사용한다.

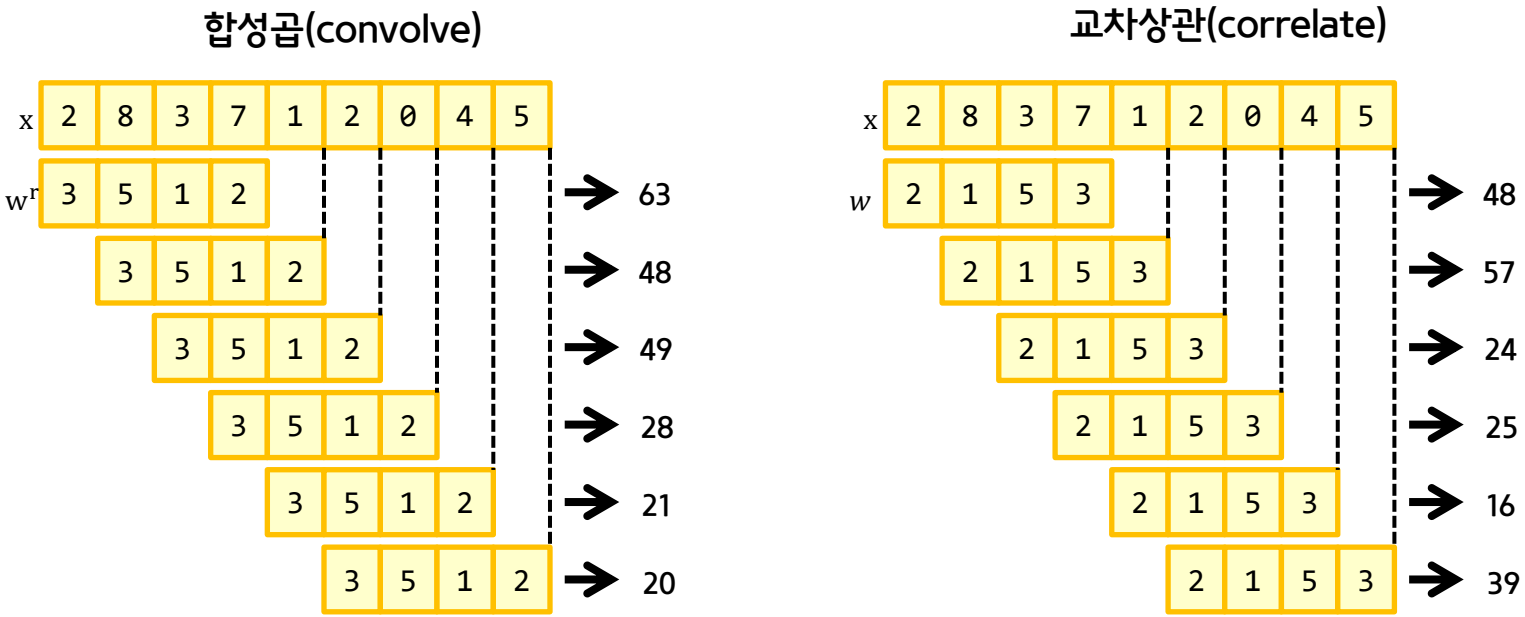


싸이파이로 교차상관 수행

```
from scipy.signal import correlate
correlate(x, w, mode='valid')
```

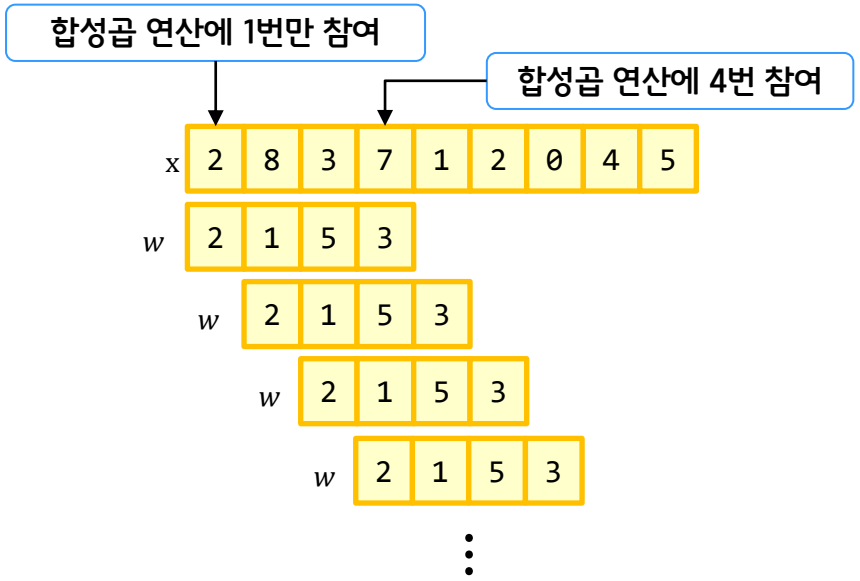
array([48, 57, 24, 25, 16, 39])

합성곱 신경망은 진짜 합성곱을 사용하지 않는다.
합성곱 대신 교차상관을 사용한다.



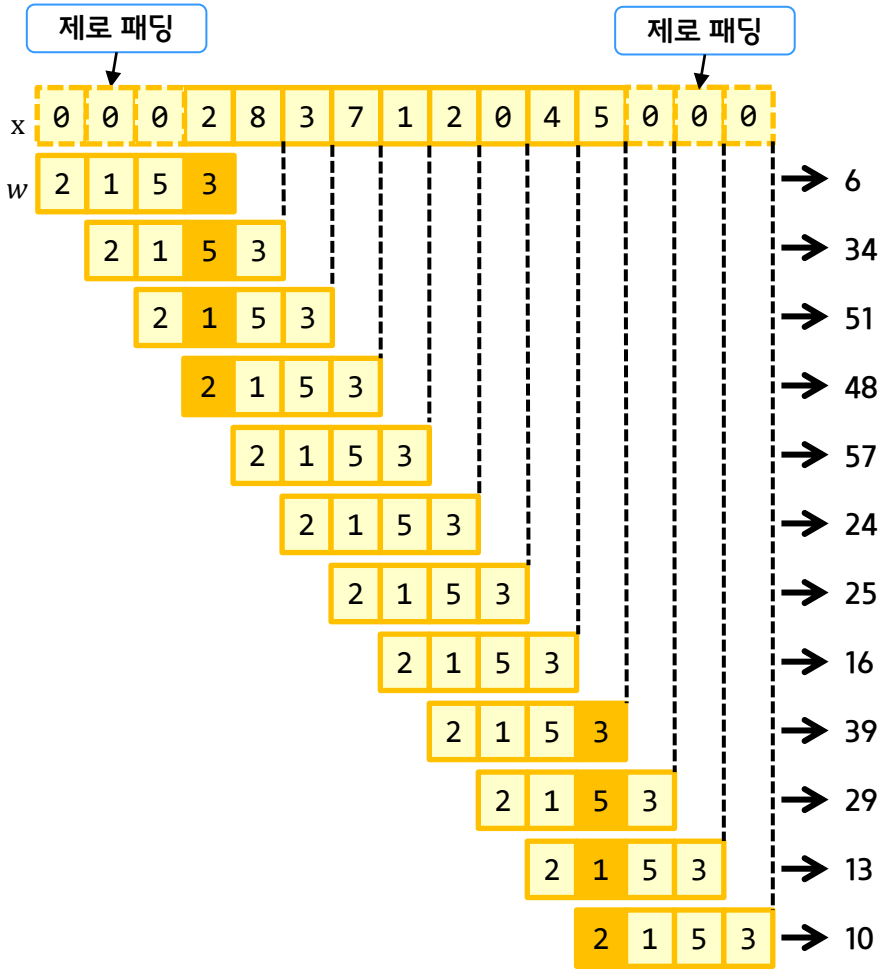
패딩과 스트라이드 이해

밸리드 패딩은 원본 배열의 원소가 합성곱 연산에 참여하는 정도가 다르다.



패딩과 스트라이드 이해

풀 패딩은 원본 배열의 원소의 연산 참여도를 동일하게 만든다.

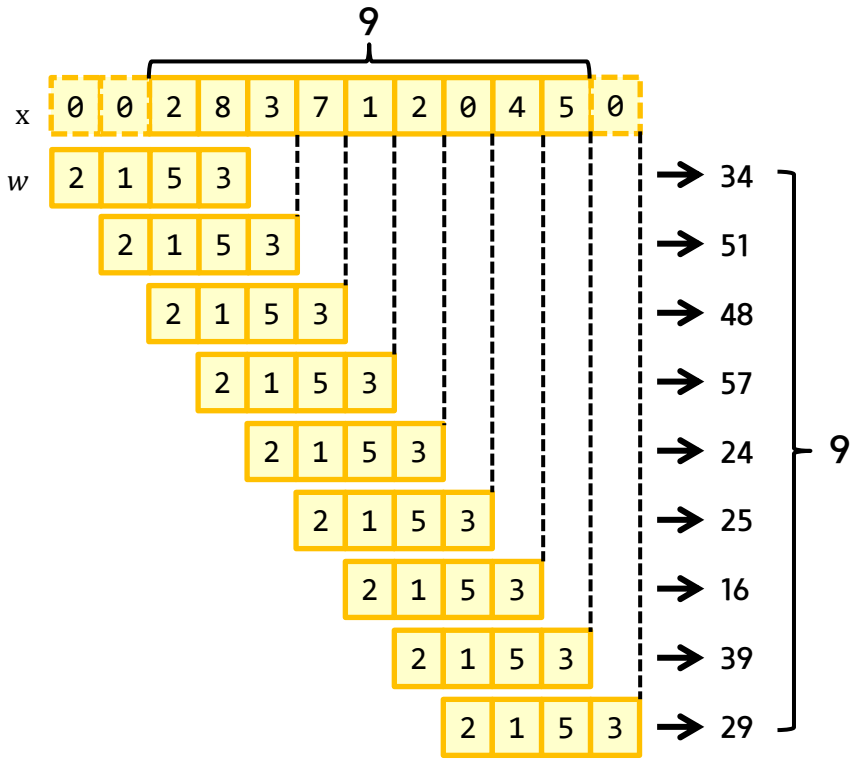



```
correlate(x, w, mode='full')
```

```
array([ 6, 34, 51, 48, 57, 24, 25, 16, 39, 29, 13, 10])
```

패딩과 스트라이드 이해

세임 패딩은 출력 배열의 길이를 원본 배열의 원소의 길이와 동일하게 만든다.

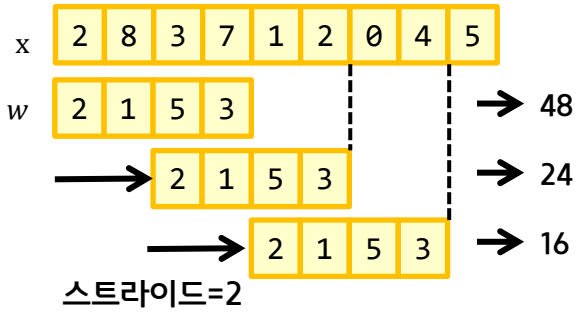



```
correlate(x, w, mode='same')
```

```
array([34, 51, 48, 57, 24, 25, 16, 39, 29])
```

패딩과 스트라이드 이해

스트라이드는 미끄러지는 간격을 조정한다.




```
correlate(x, w, mode='same')
```

```
array([34, 51, 48, 57, 24, 25, 16, 39, 29])
```

2차원 배열에서 합성곱 수행 (mode='valid')

x

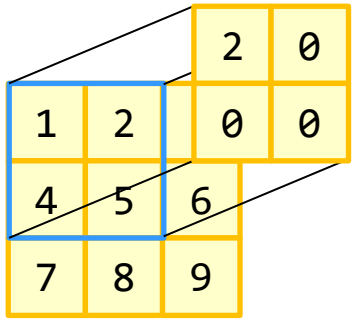
1	2	3
4	5	6
7	8	9

w

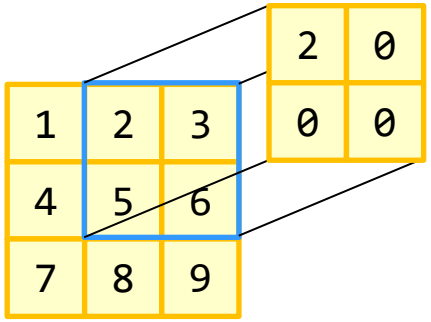
2	0
0	0

```
x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
w = np.array([[2, 0], [0, 0]])
from scipy.signal import correlate2d
correlate2d(x, w, mode='valid')
```

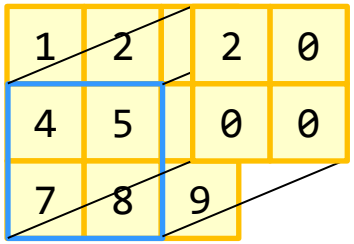
```
array([[ 2,  4],
       [ 8, 10]])
```



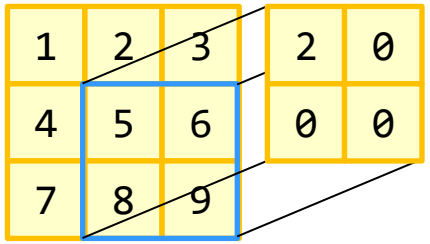
=> 2



=> 4

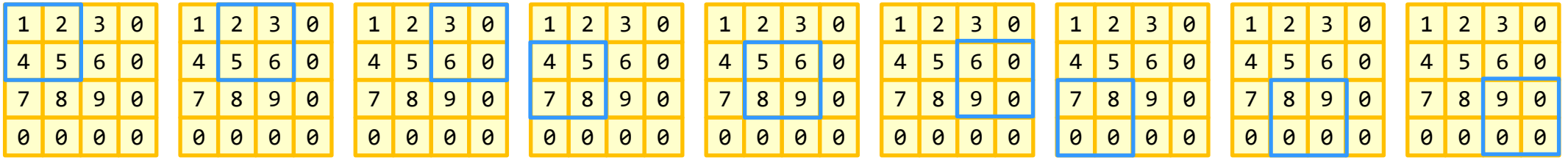


=> 8



=> 10

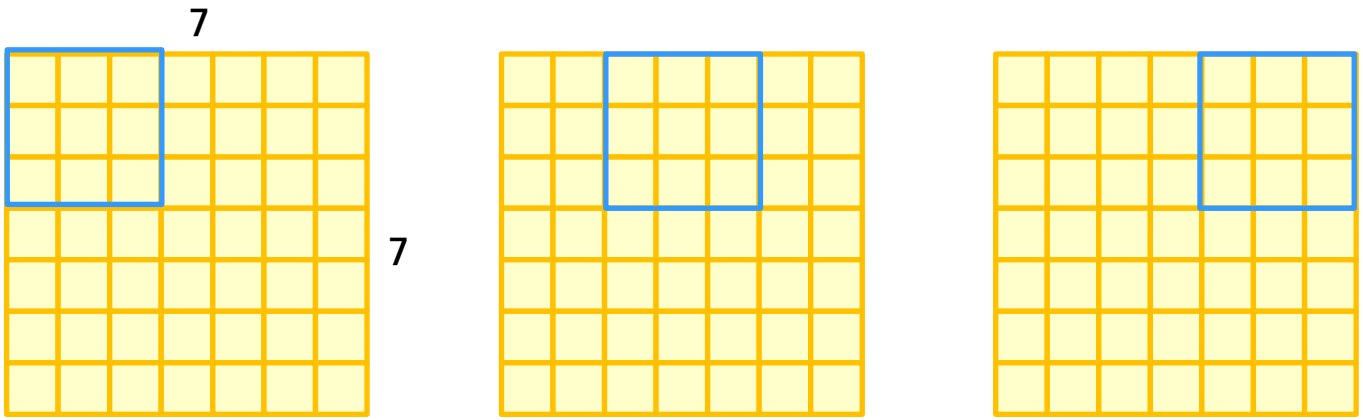
2차원 배열에서 same padding




```
correlate2d(x, w, mode='same')
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

2차원 배열에서 스트라이드 이해



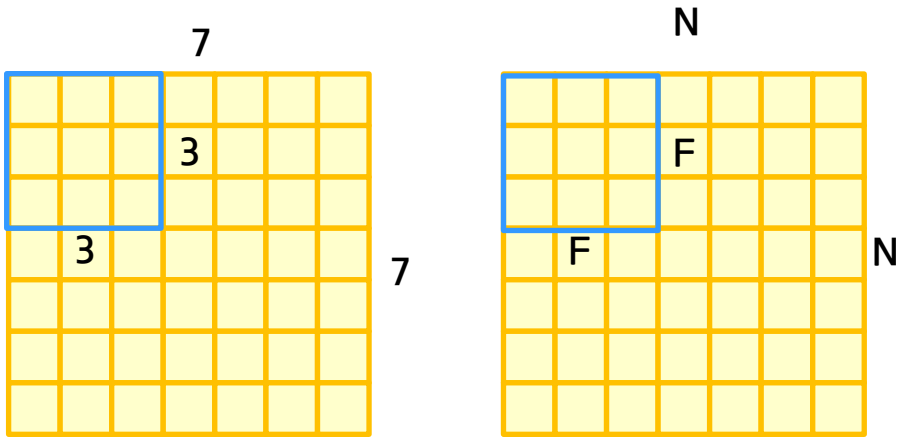
7x7 input
3x3 filter
mode = 'valid'
stride = 1

=> 5x5 output

7x7 input
3x3 filter
mode = 'valid'
stride = 2

=> 3x3 output

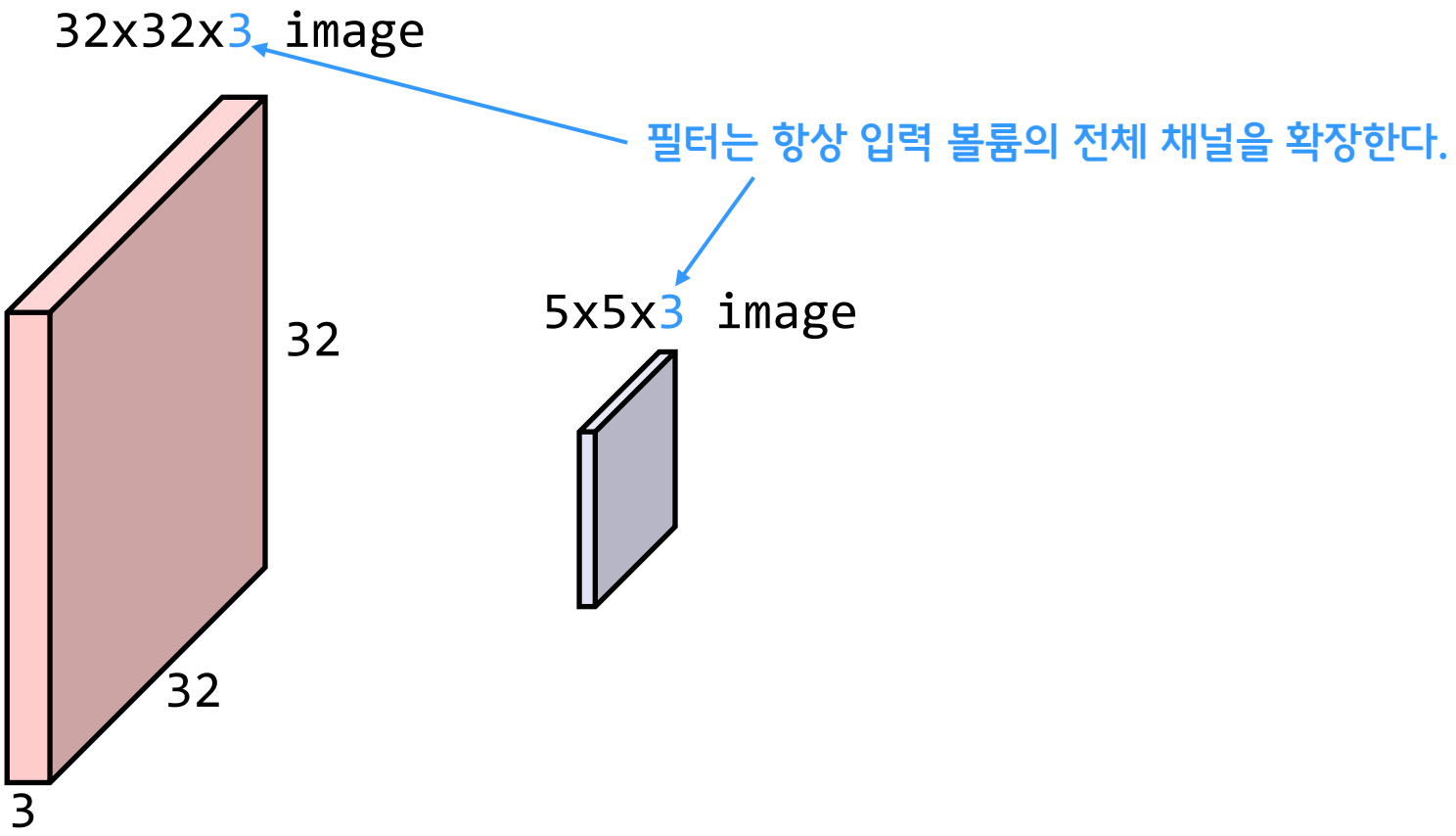
2차원 배열에서 스트라이드 이해



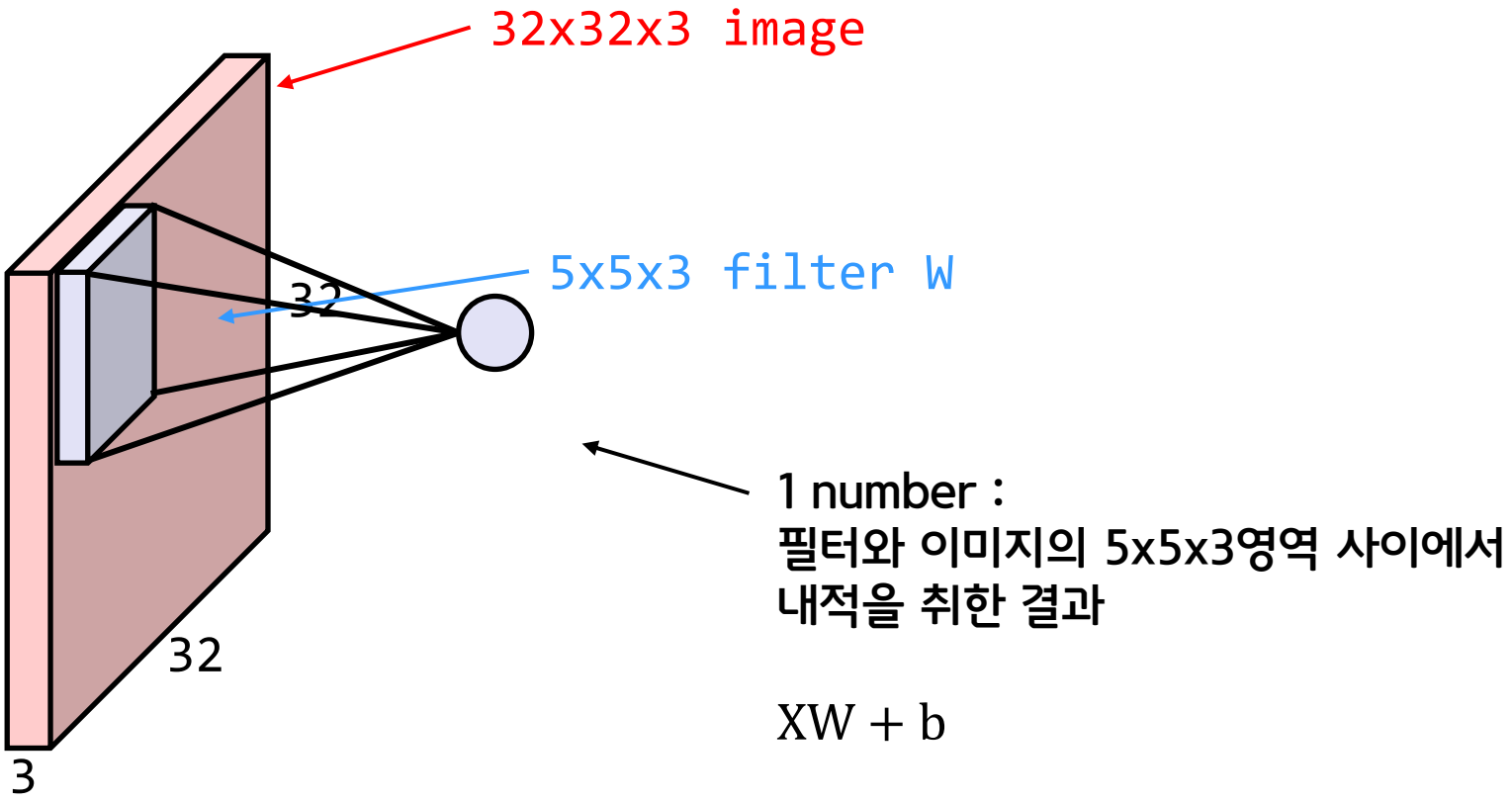
Output size :
(N - F) / stride + 1

예) N = 7, F = 3
stride 1 => (7-3)/1+1 = 5
stride 2 => (7-3)/2+1 = 3
stride 3 => (7-3)/3+1 = 2.33

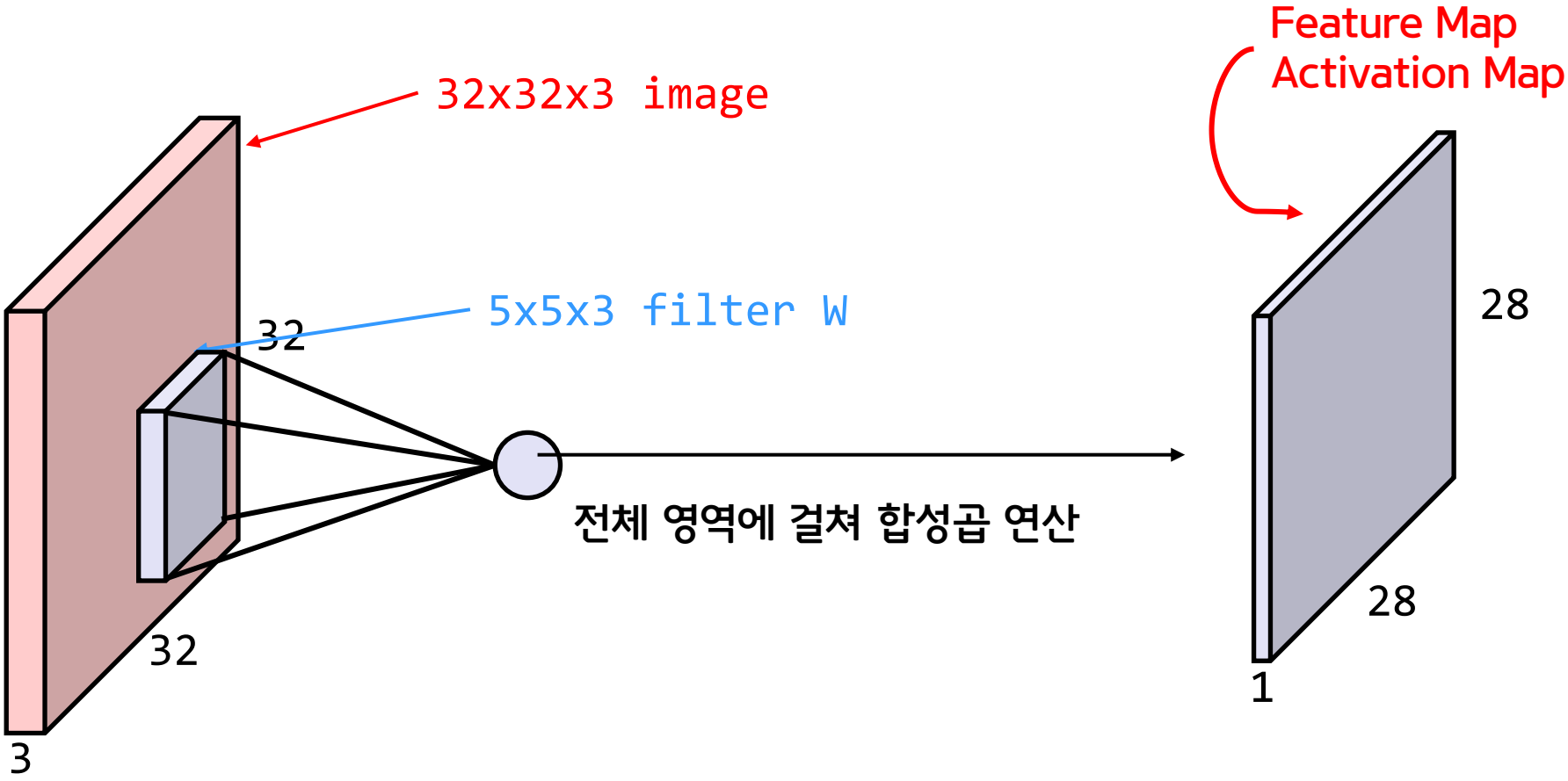
Convolution Layer

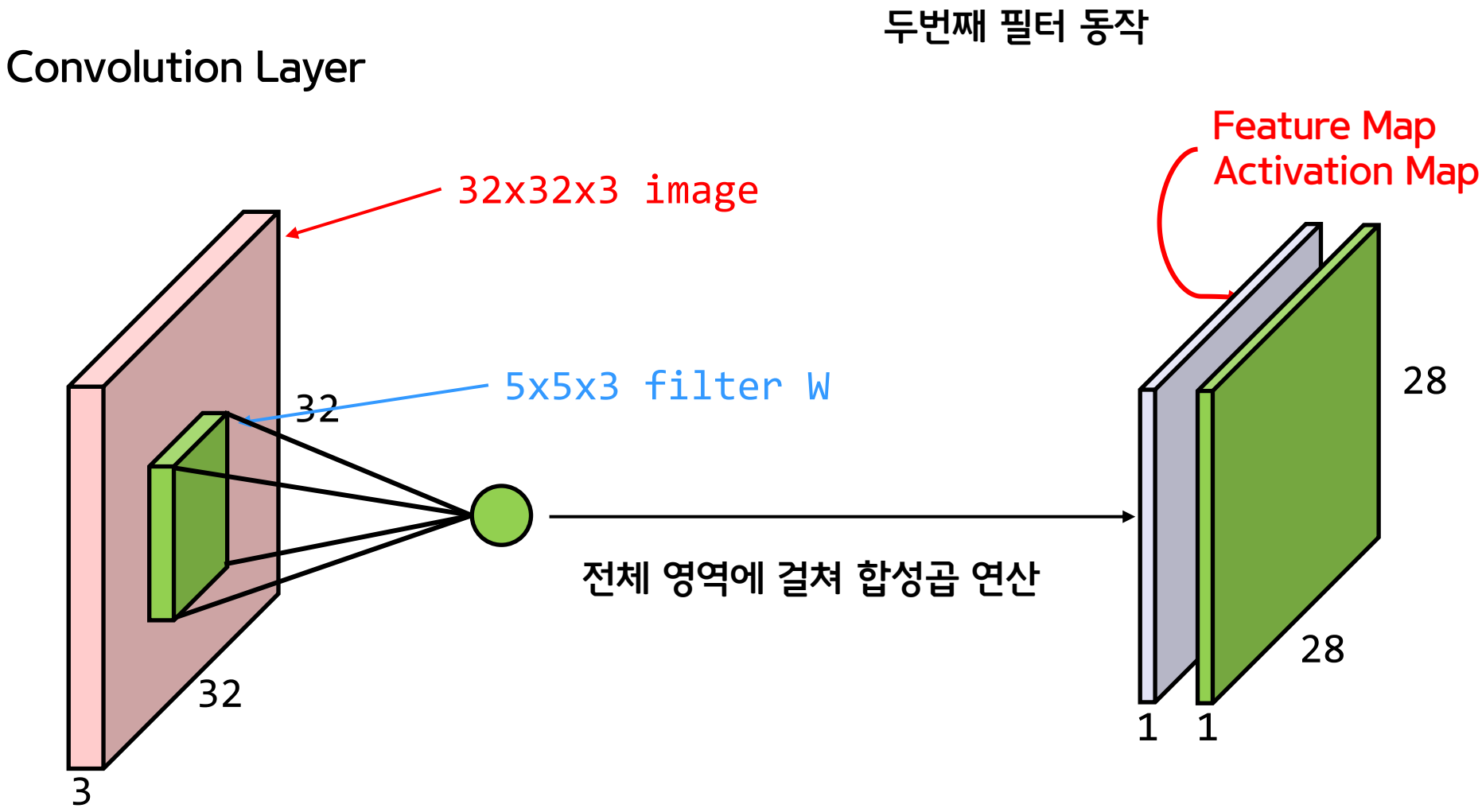


Convolution Layer



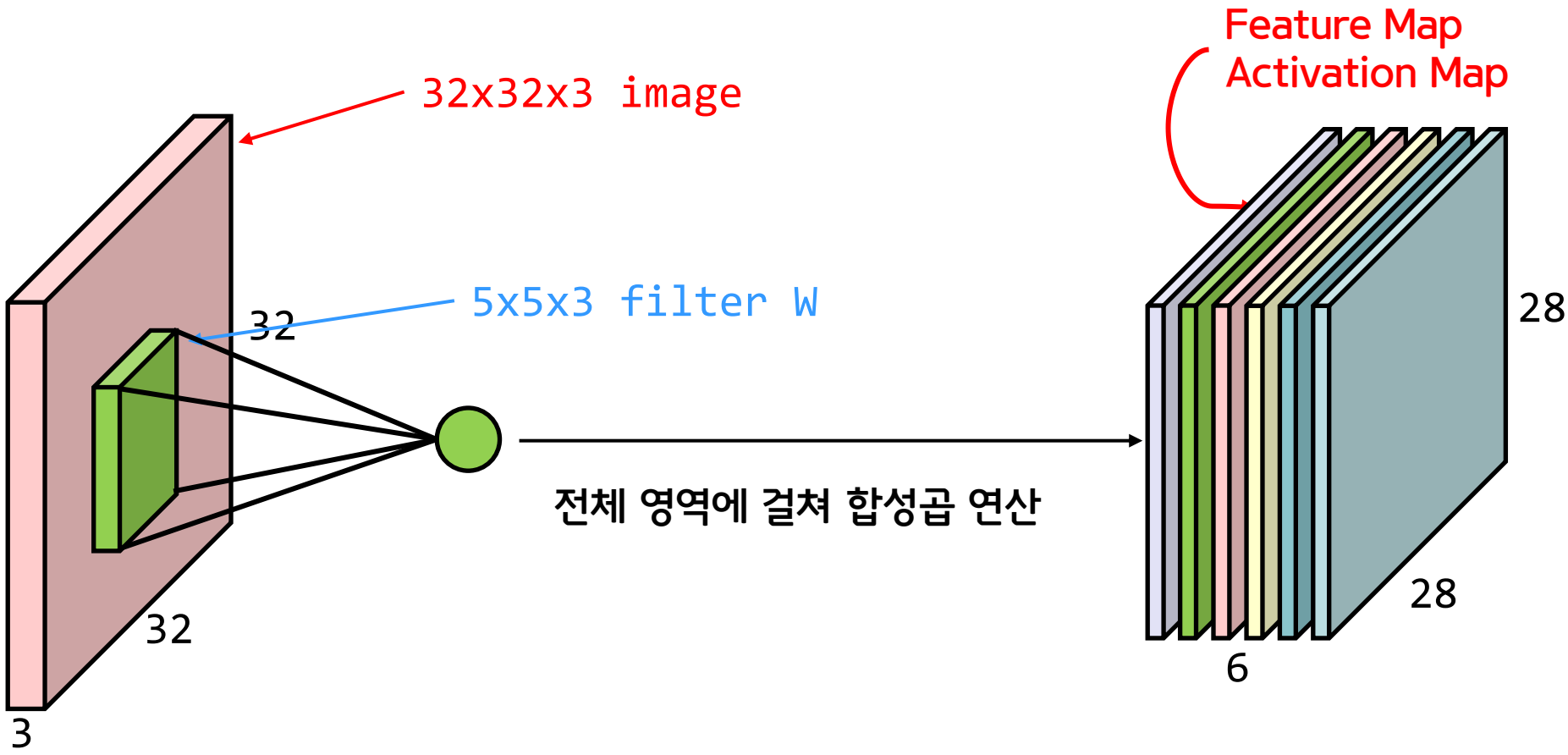
Convolution Layer




두번째 필터 동작
Convolution Layer
Feature Map
Activation Map
32x32x3 image
5x5x3 filter W
32
32
3
전체 영역에 걸쳐 합성곱 연산
28
28
1
1

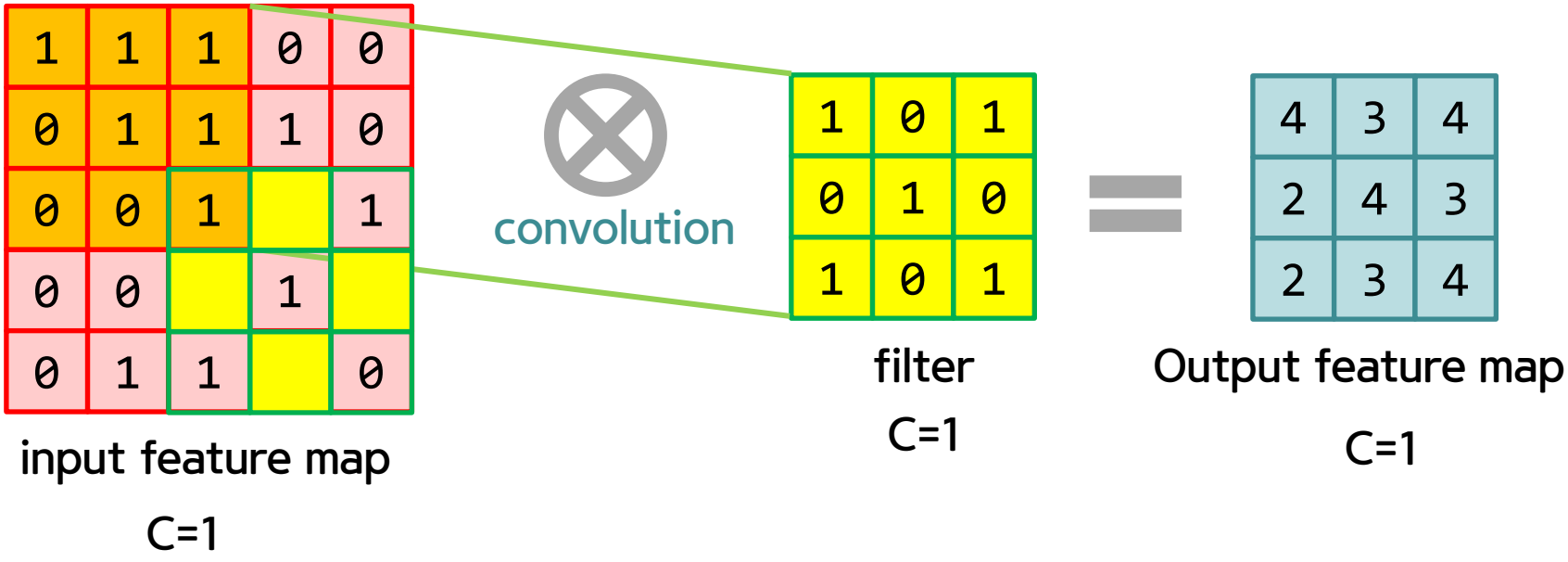
Convolution Layer

6개의 5x5필터가 있다면, 6개의 개별 feature map이 생성됨

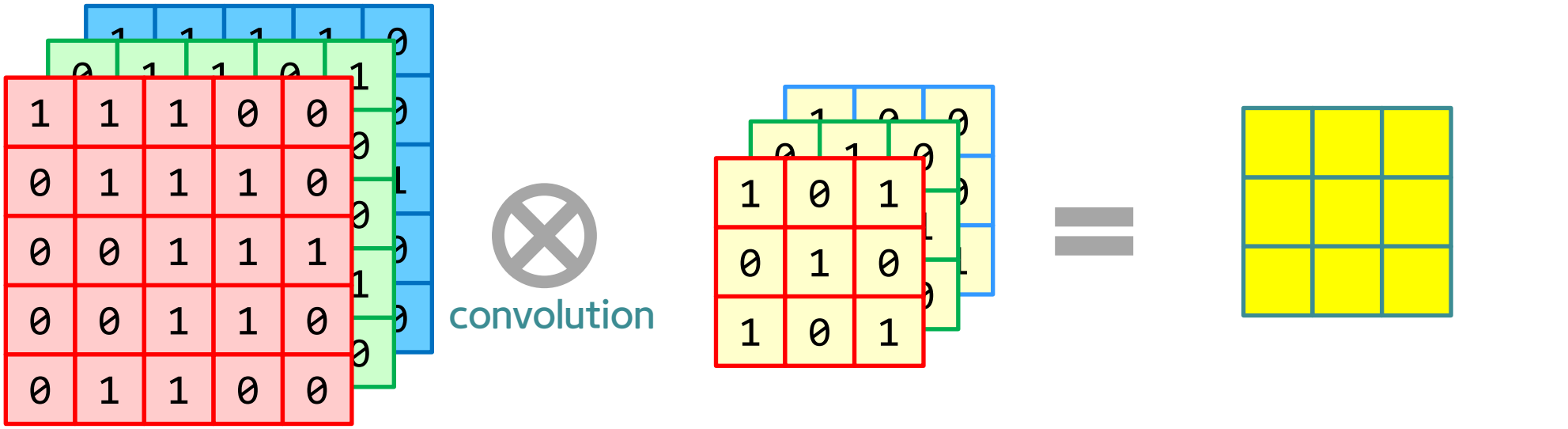


Convolution Layer – 계산

1x1 + 1x0 + 1x1 + 0x0 + 1x1 + 1x0 + 0x1 + 0x0 +1x1 = 4



Convolution Layer – 계산



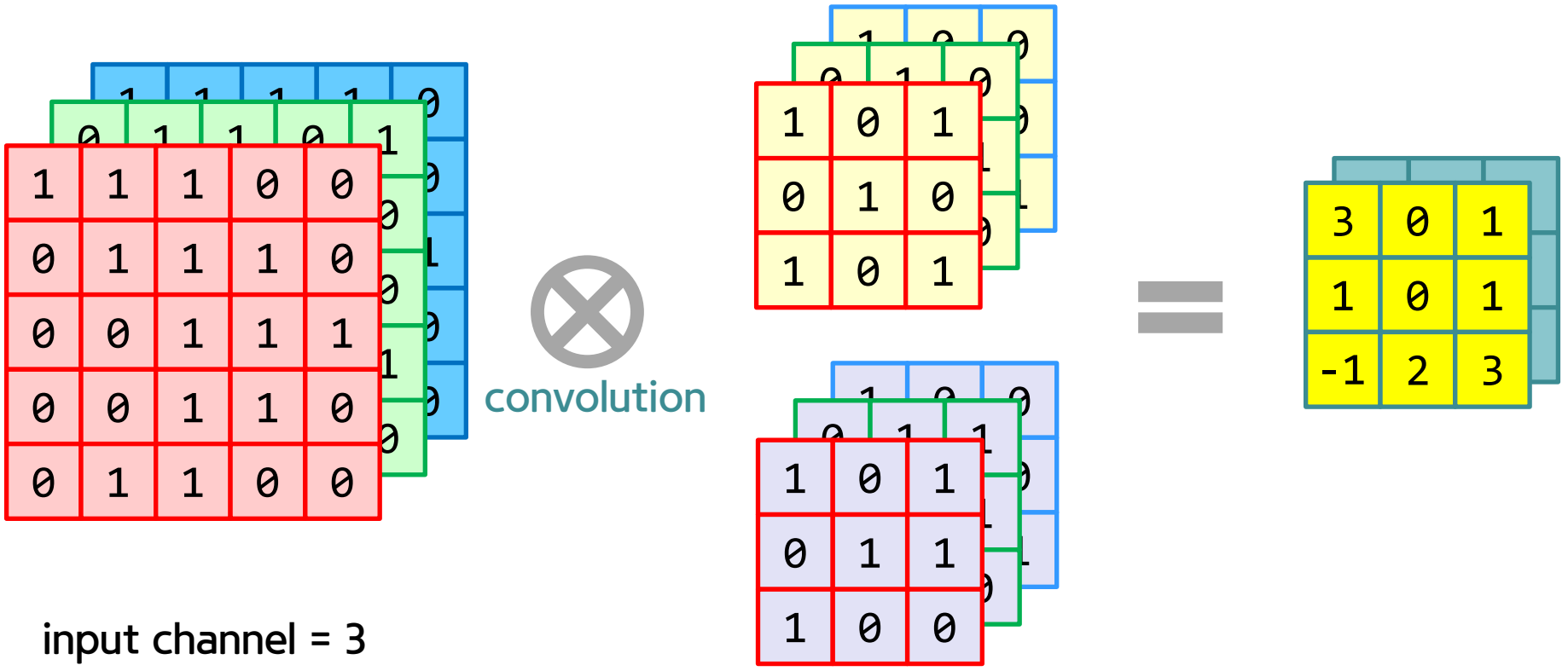
input channel = 3

filter channel = 3

# of filters : 1

Output channel : 1

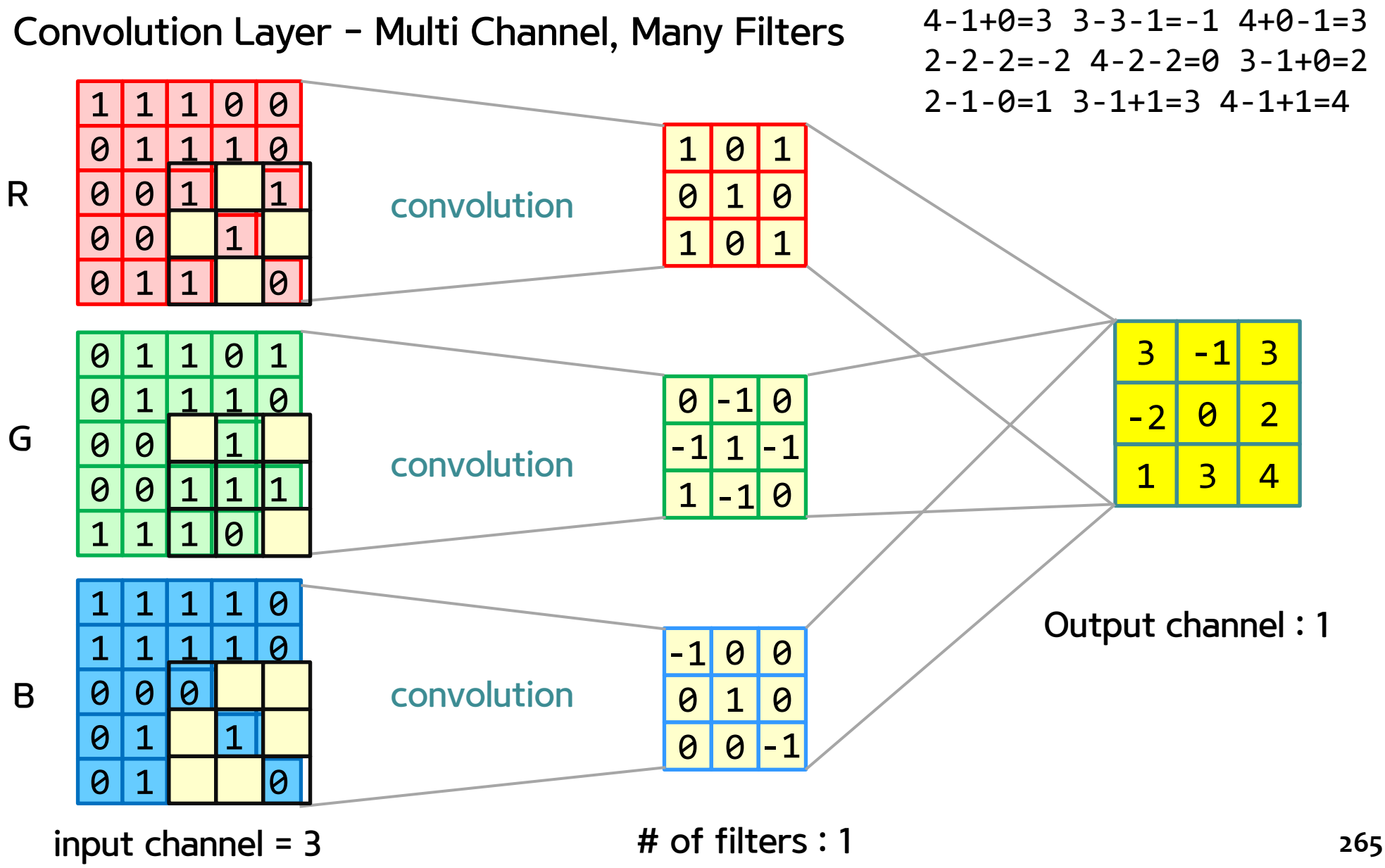
Convolution Layer – 계산

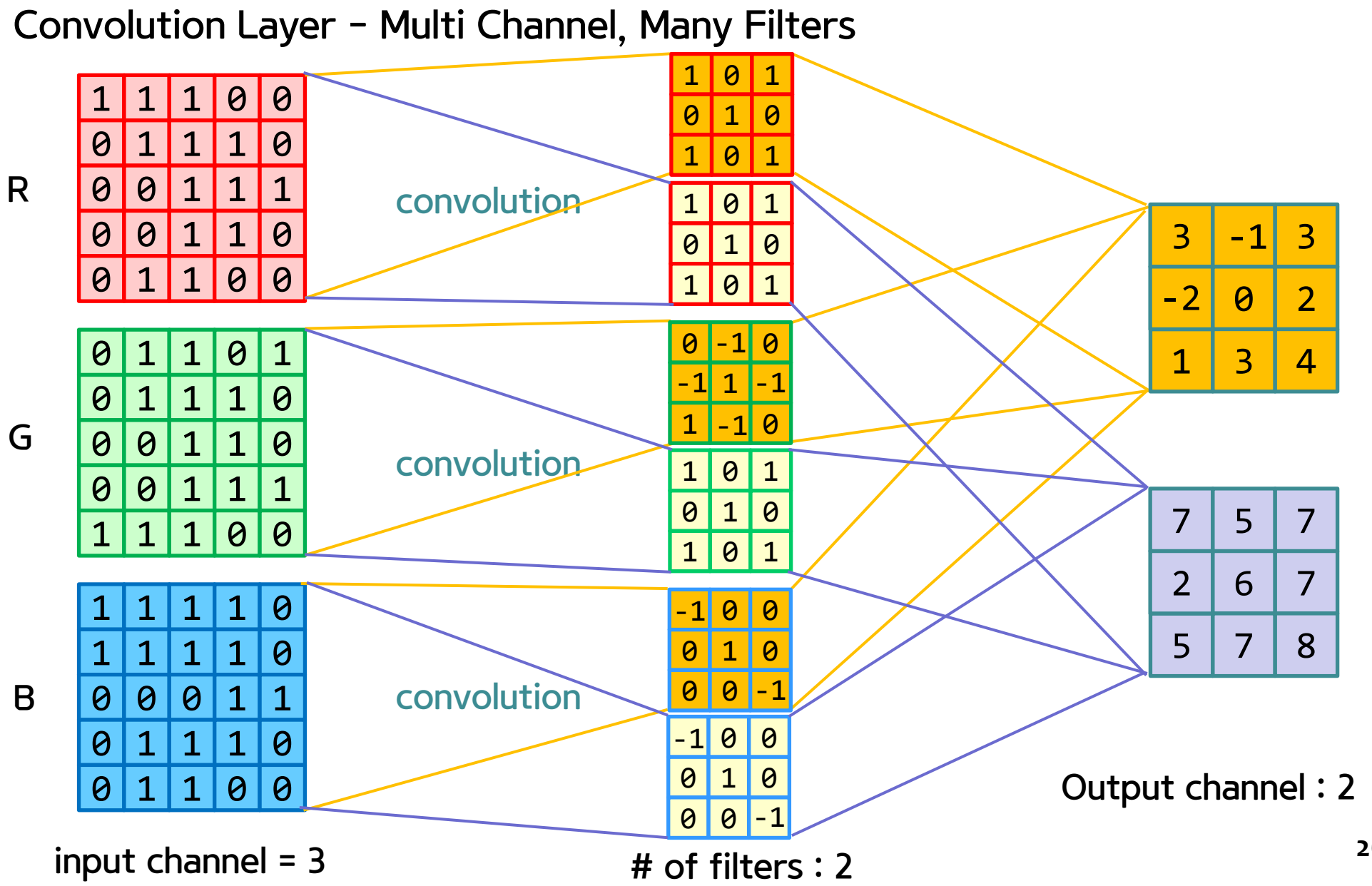


input channel = 3

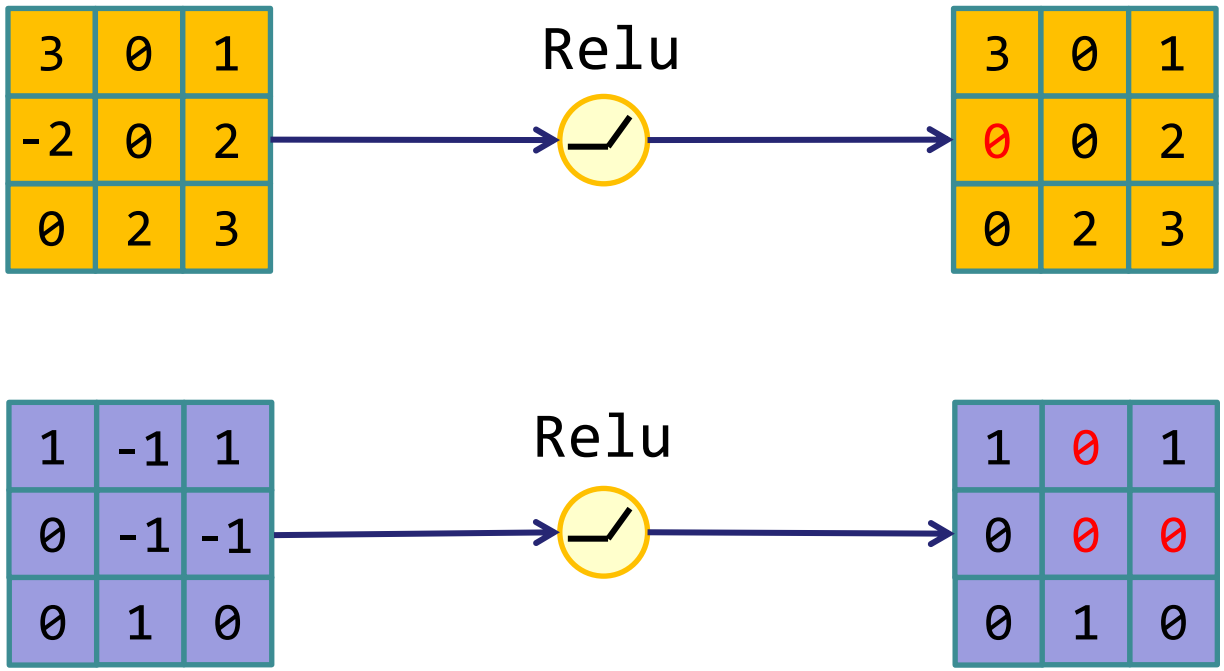
filter channel = 3
# of filters : 2

Output channel : 2


Convolution Layer - Multi Channel, Many Filters
4-1+0=3 3-3-1=-1 4+0-1=3
2-2-2=-2 4-2-2=0 3-1+0=2
2-1-0=1 3-1+1=3 4-1+1=4
R
G
B
convolution
convolution
convolution
Output channel : 1
input channel = 3
# of filters : 1


Convolution Layer - Multi Channel, Many Filters
R
G
B
convolution
convolution
convolution
input channel = 3
# of filters : 2
Output channel : 2

Activation Function



tf.keras.layers.Conv2D

Arguments:

- **`filters`**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

- **`kernel_size`**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

- **`strides`**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.

- **`padding`**: one of `"valid"` or `"same"` (case-insensitive).

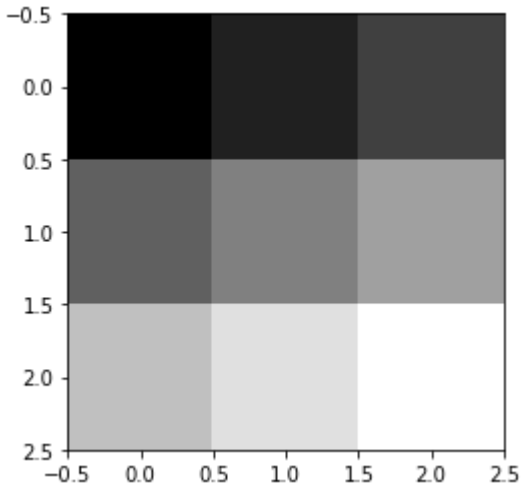
- **`data_format`**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch_size, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch_size, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

tf.keras.layers.Conv2D

- **activation**: Activation function to use. If you don't specify anything, no activation is applied (see keras.activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix (see keras.initializers).
- **bias_initializer**: Initializer for the bias vector (see keras.initializers).
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix (see keras.regularizers).
- **bias_regularizer**: Regularizer function applied to the bias vector (see keras.regularizers).

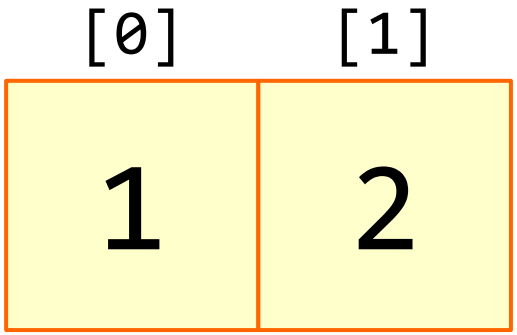
```
import tensorflow as tf
import numpy as np
import keras
from keras.layers import *
import matplotlib.pyplot as plt
image = tf.constant([[[[1],[2],[3]],
                      [[4],[5],[6]],
                      [[7],[8],[9]]]], dtype=np.float32)
print(image.shape)
plt.imshow(image.numpy().reshape(3,3), cmap='gray')
```

```
(1, 3, 3, 1)
```



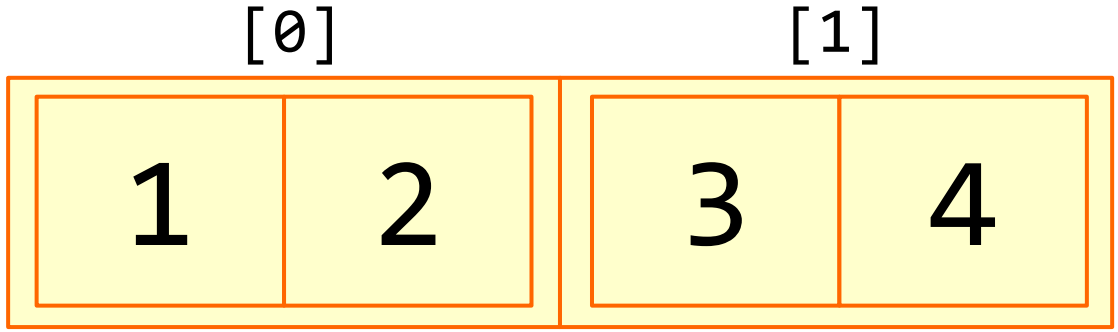
1차원 배열

```
a = np.array( [1, 2] )
```



2차원 배열

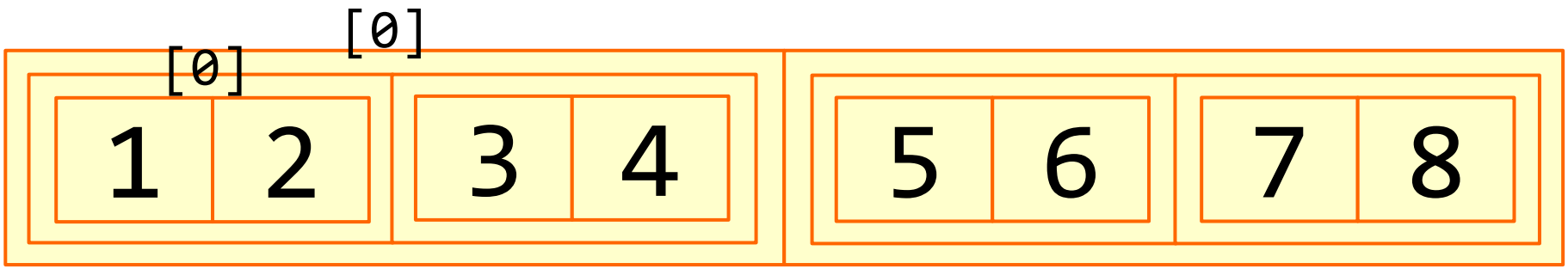
```
a = np.array( [ [1, 2], [3, 4] ] )
len(a)
len(a[0])
len(a[1])
```



	[0]	[1]
[0]	1	2
[1]	3	4

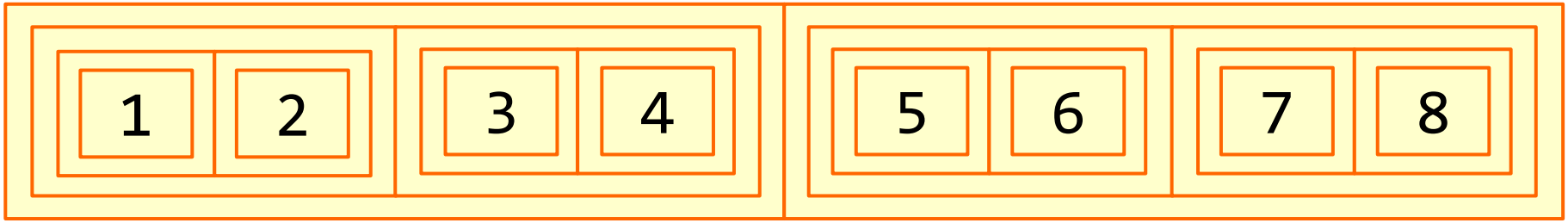
3차원 배열

```
a = np.array( [ [ [1,2],[3,4] ], [[5,6],[7,8]] ] )
len(a)
len(a[0])
len(a[1])
len(a[0][0])
```

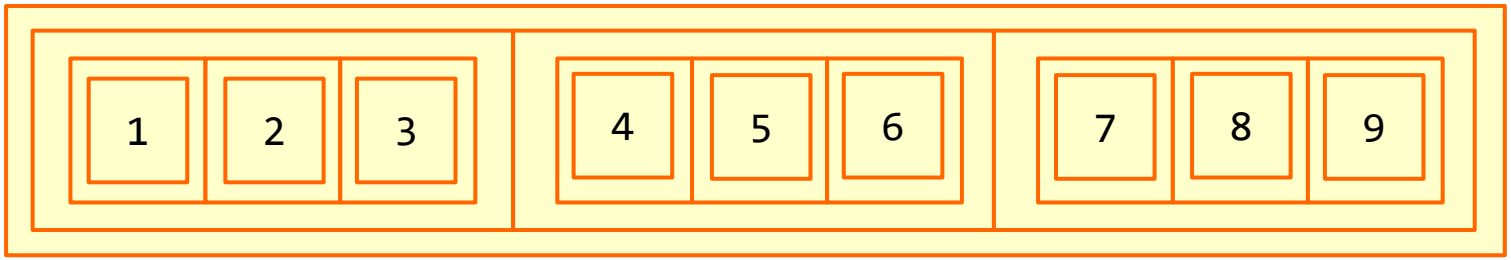


4차원 배열

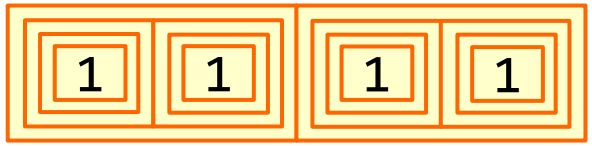
```
a = np.array( [ [ [ [1], [2] ], [[3],[4]] ],
                [[[5],[6]],[[7],[8]]]  ] )
len(a)
len(a[0])
len(a[0][0][0])
```



```
[       [ [ [1],[2],[3] ],
          [[4],[5],[6]],
          [[7],[8],[9]]  ]  ]

len(a)
len(a[0])
len(a[0][0])
len(a[0][0][0])
```



```
weight = np.array([  [ [ [1.] ], [[1.]] ]  ,  [[[1.]],[[1.]]]   ])
print("weight.shape=", weight.shape)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
print(conv2d.numpy().reshape(3,3))
plt.imshow(conv2d.numpy().reshape(3,3), cmap='gray')
plt.show()
```

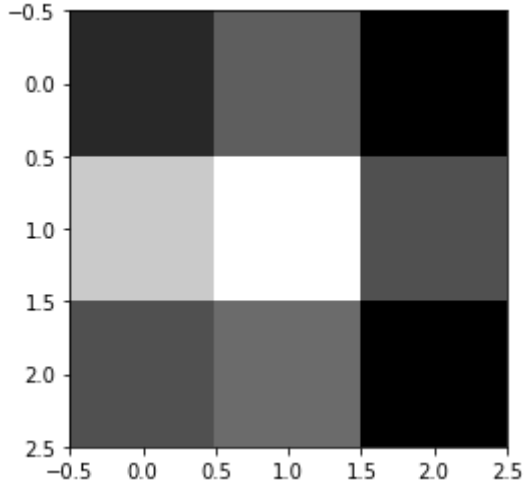


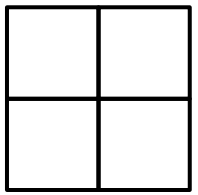
(2,2,1,1)

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

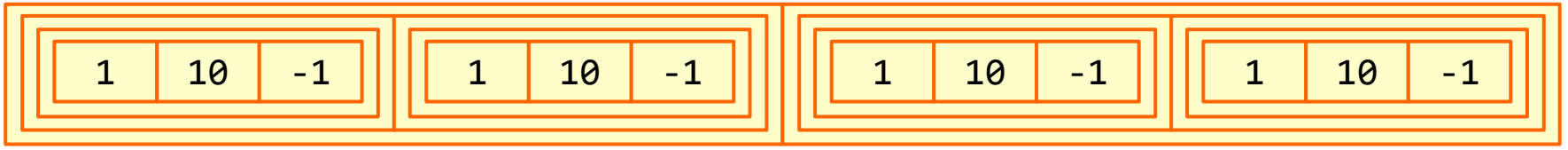
1	1
1	1

```
conv2d.shape (1, 3, 3, 1)
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
```





```
weight = np.array([  [[[1.,10.,-1.]], [[1.,10.,-1.]]],  [[[1.,10.,-1.]],[[1.,10.,-1.]]]  ])
```




```
weight.shape (2, 2, 1, 3)
conv2d.shape (1, 3, 3, 1)
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
```

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

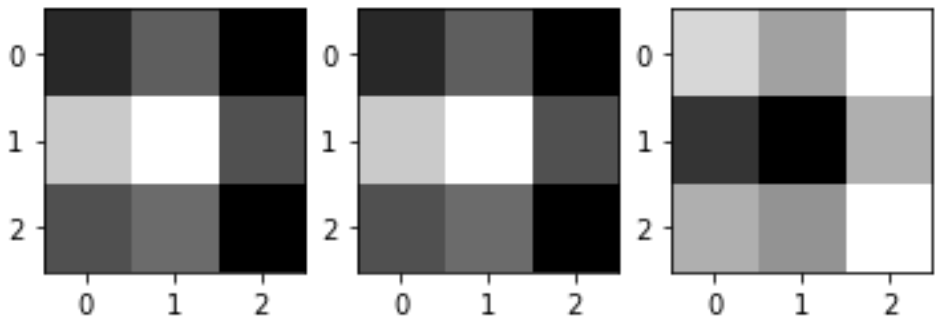
1	1
1	1

10	10
10	10

-1	-1
-1	-1

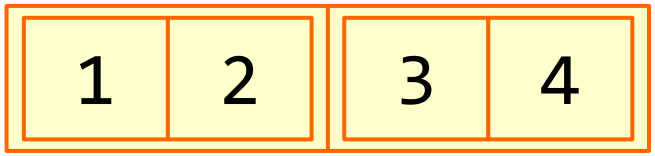
```
print("image.shpe", image.shape)
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]]],[[[1.,10.,-1.]],[[1.,10.,-1.]]]])
print("weight.shpe", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=3, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
feature_maps = np.swapaxes(conv2d, 0, 3)
for i, feature_map in enumerate(feature_maps):
    print(feature_map.reshape(3,3))
    plt.subplot(1,3,i+1), plt.imshow(feature_map.reshape(3,3), cmap='gray')
plt.show()
```

```
image.shape (1, 3, 3, 1)
weight.shape (2, 2, 1, 3)
conv2d.shape (1, 3, 3, 3)
[[12. 16.  9.]
 [24. 28. 15.]
 [15. 17.  9.]]
[[120. 160.  90.]
 [240. 280. 150.]
 [150. 170.  90.]]
[[-12. -16.  -9.]
 [-24. -28. -15.]
 [-15. -17.  -9.]]
```



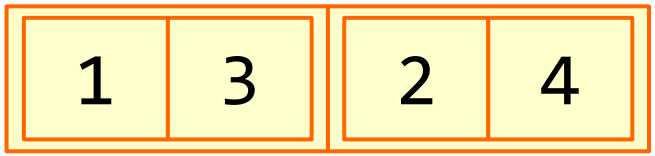
2차원 배열의 전치

```
a = np.array([[1,2],[3,4]])
```



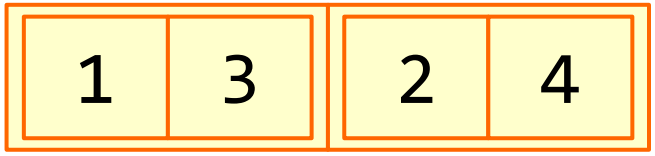
(2,2)

```
np.swapaxes(a, 0, 1)
```



(2,2)

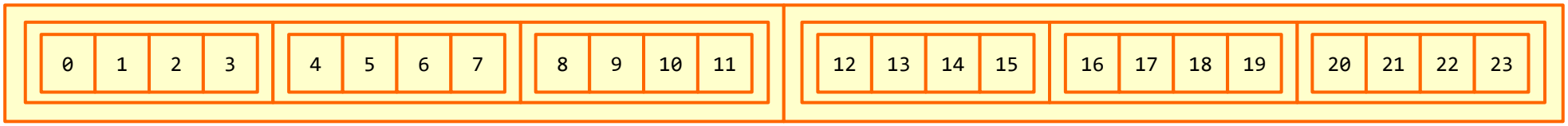
```
a.T
```



3차원 배열의 전치

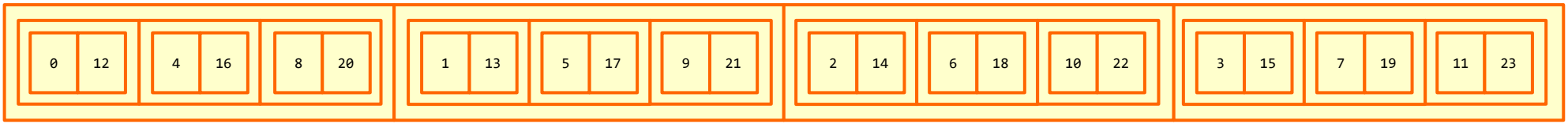
```
a = np.arange(24).reshape(2,3,4)
```

(2, 3, 4)



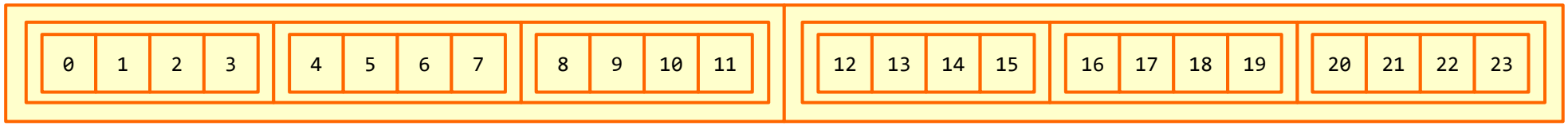

```
np.swapaxes(a, 0, 2)
```

(4, 3, 2)



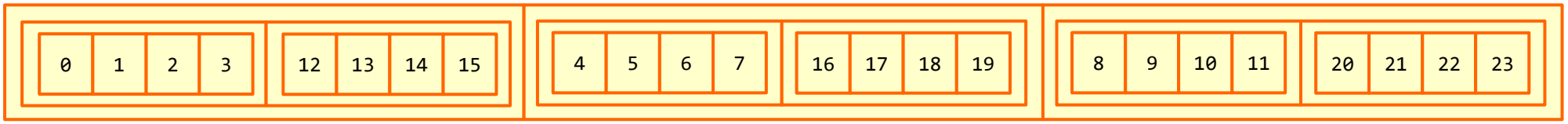
3차원 배열의 전치

```
a = np.arange(24).reshape(2,3,4)
```




```
np.swapaxes(a, 0, 1)
```

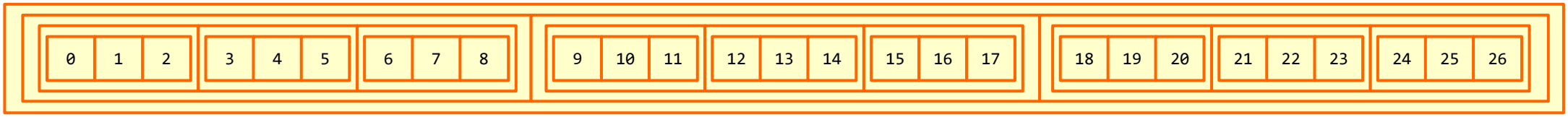
```
(3, 2, 4)
```



4차원 배열의 전치

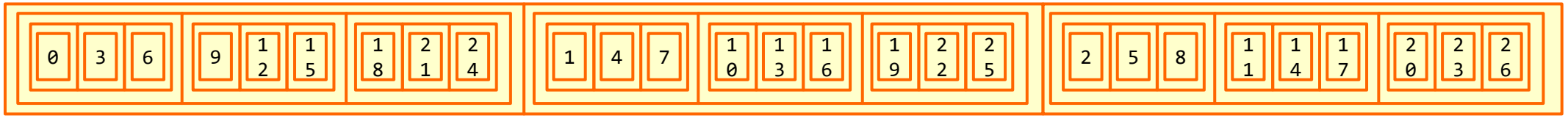
```
a = np.arange(27).reshape(1,3,3,3)
```

(1, 3, 3, 3)




```
np.swapaxes(a, 0, 3)
```

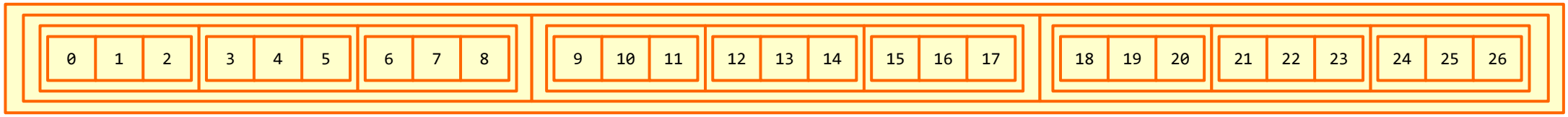
(3, 3, 3, 1)



4차원 배열의 전치

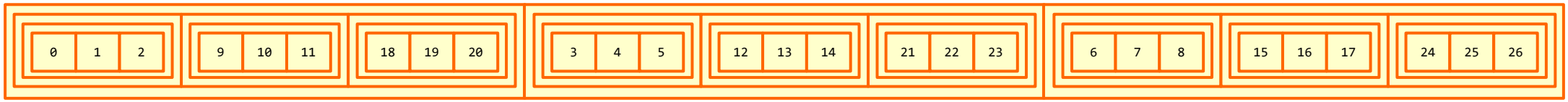
```
a = np.arange(27).reshape(1,3,3,3)
```

(1, 3, 3, 3)




```
np.swapaxes(a, 0, 2)
```

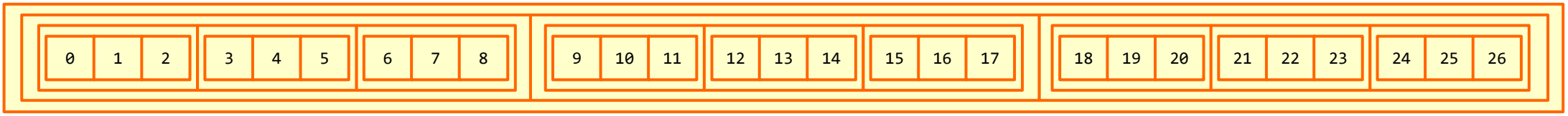
(3, 3, 1, 3)



4차원 배열의 전치

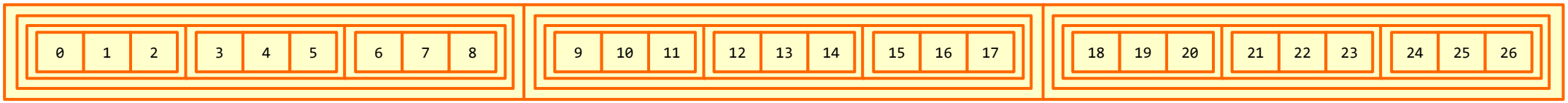
```
a = np.arange(27).reshape(1,3,3,3)
```

```
(1, 3, 3, 3)
```




```
np.swapaxes(a, 0, 1)
```

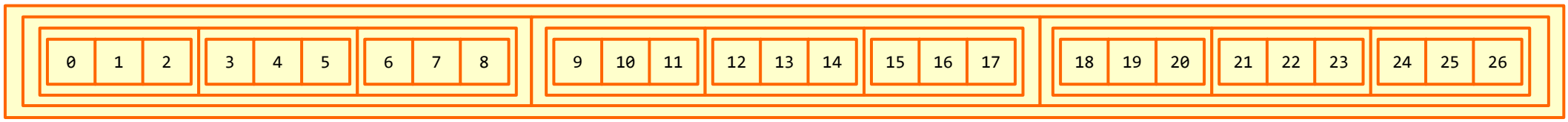
```
(3, 1, 3, 3)
```



4차원 배열의 전치

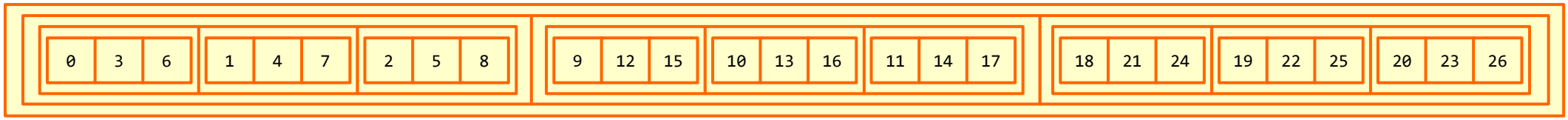
```
a = np.arange(27).reshape(1,3,3,3)
```

```
(1, 3, 3, 3)
```




```
np.swapaxes(a, 2, 3)
```

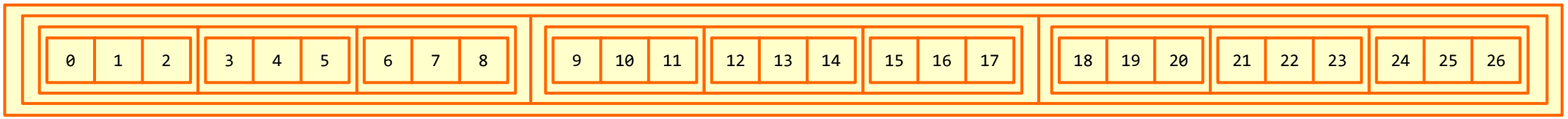
```
(1, 3, 3, 3)
```



4차원 배열의 전치

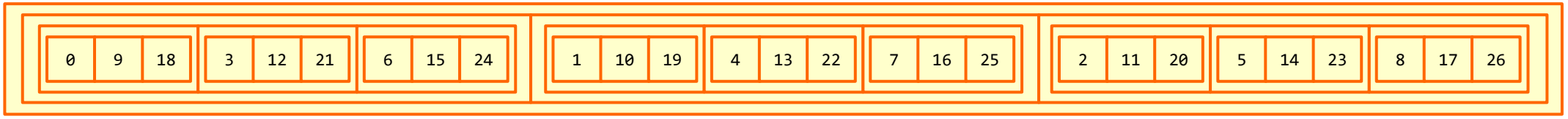
```
a = np.arange(27).reshape(1,3,3,3)
```

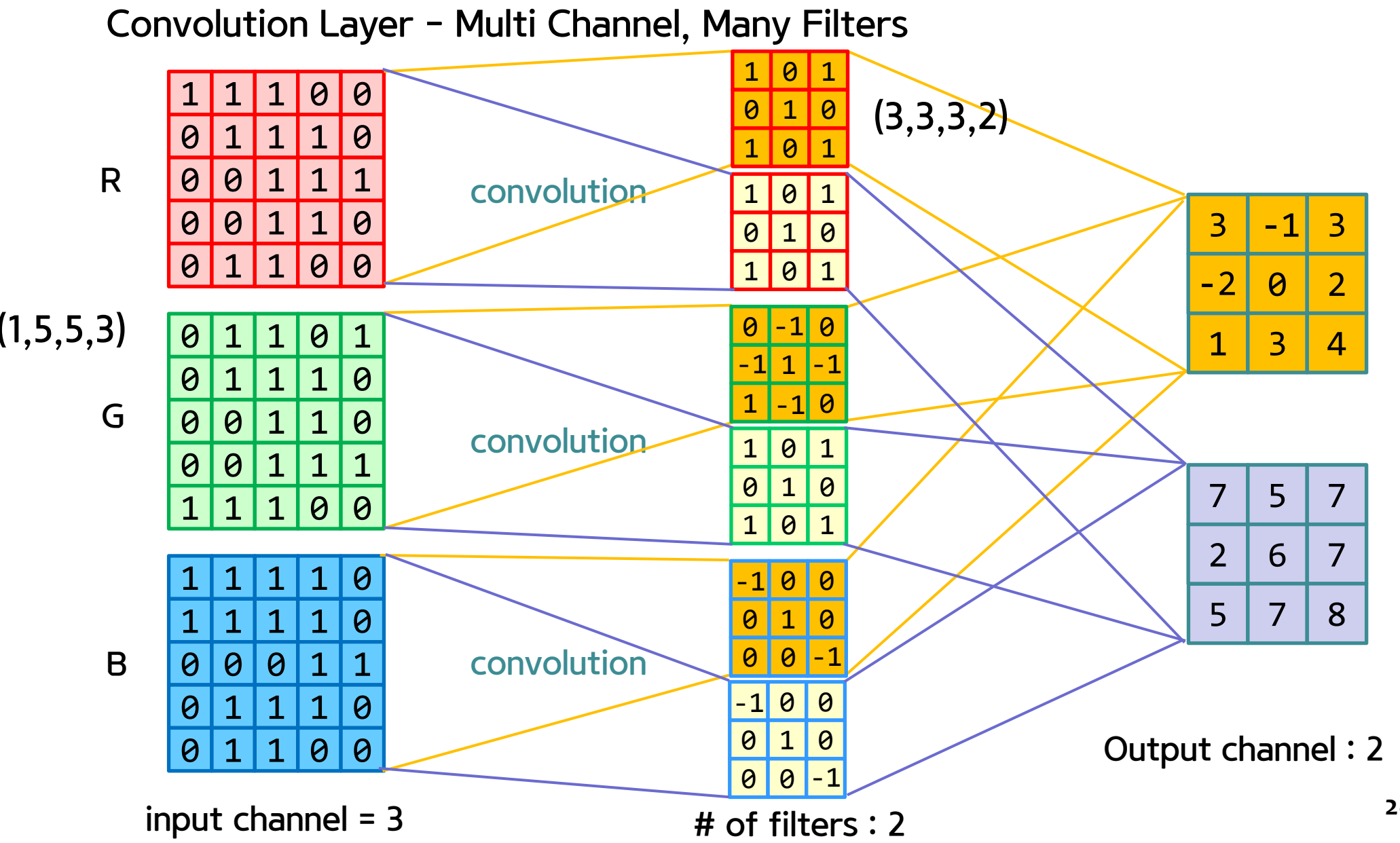
(1, 3, 3, 3)




```
np.swapaxes(a, 1, 3)
```

(1, 3, 3, 3)




Convolution Layer - Multi Channel, Many Filters
R
G
B
(1,5,5,3)
convolution
convolution
convolution
(3,3,3,2)
input channel = 3
# of filters : 2
Output channel : 2

```
image = tf.constant(  [[
                          [[1,0,1],[1,1,1],[1,1,1],[0,0,1],[0,1,0]],
                          [[0,0,1],[1,1,1],[1,1,1],[1,1,1],[0,0,0]],
                          [[0,0,0],[0,0,0],[1,1,0],[1,1,1],[1,0,1]],
                          [[0,0,0],[0,0,1],[1,1,1],[1,1,1],[0,1,0]],
                          [[0,1,0],[1,1,1],[1,1,1],[0,0,0],[0,0,0]]
                       ]],     dtype=np.float32)
```

```
[[1. 1. 1. 0. 0.]
 [0. 1. 1. 1. 0.]
 [0. 0. 1. 1. 1.]
 [0. 0. 1. 1. 0.]
 [0. 1. 1. 0. 0.]]
[[0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 0.]
 [0. 0. 1. 1. 0.]
 [0. 0. 1. 1. 1.]
 [1. 1. 1. 0. 0.]]
[[1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 0.]
 [0. 0. 0. 1. 1.]
 [0. 1. 1. 1. 0.]
 [0. 1. 1. 0. 0.]]
```

(1,5,5,3)

R

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

G

0	1	1	0	1
0	1	1	1	0
0	0	1	1	0
0	0	1	1	1
1	1	1	0	0

B

1	1	1	1	0
1	1	1	1	0
0	0	0	1	1
0	1	1	1	0
0	1	1	0	0

input channel = 3

```
image = tf.constant(  [[
                          [[1,0,1],[1,1,1],[1,1,1],[0,0,1],[0,1,0]],
                          [[0,0,1],[1,1,1],[1,1,1],[1,1,1],[0,0,0]],
                          [[0,0,0],[0,0,0],[1,1,0],[1,1,1],[1,0,1]],
                          [[0,0,0],[0,0,1],[1,1,1],[1,1,1],[0,1,0]],
                          [[0,1,0],[1,1,1],[1,1,1],[0,0,0],[0,0,0]]
                       ]],     dtype=np.float32)

maps = np.swapaxes(image, 0, 3)
for i, map in enumerate(maps):
    print(map.reshape(5,5))
```

```
weight = np.array( [
                     [[1],[0],[-1]], [[0],[-1],[0]], [[1],[0],[0]]],
                     [[0],[-1],[0]], [[1],[1],[1]], [[0],[-1],[0]]],
                     [[1],[1],[0]], [[0],[-1],[0]], [[1],[0],[-1]]]
                   ] )
maps = np.swapaxes(weight, 1, 2)
maps = np.swapaxes(maps, 0, 1)

for i, map in enumerate(maps):
    print(map.reshape(3,3))
```

(3,3,3,1)

(3,3,3,1)

(3,3,3,1)

```
[[1 0 1]
 [0 1 0]
 [1 0 1]]
[[ 0 -1  0]
 [-1  1 -1]
 [ 1 -1  0]]
[[-1  0  0]
 [ 0  1  0]
 [ 0  0 -1]]
```

1	0	1
0	1	0
1	0	1

0	-1	0
-1	1	-1
1	-1	0

-1	0	0
0	1	0
0	0	-1

```
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=3, padding='valid',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)
feature_maps = np.swapaxes(conv2d, 0, 3)
for i, feature_map in enumerate(feature_maps):
    print(feature_map.reshape(3,3))
```

```
(1,3,3,1)


[[ 3. -1.  3.]
 [-2.  0.  2.]
 [ 1.  3.  4.]]
```

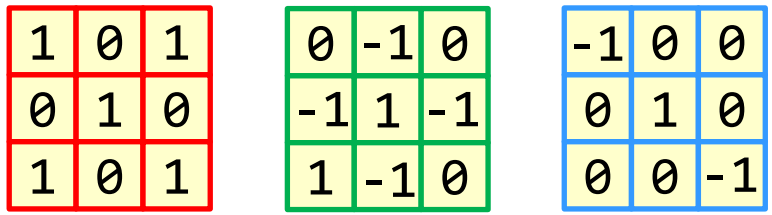
3	-1	3
-2	0	2
1	3	4

```
weight = np.array( [
                     [[1,1],[0,1],[-1,-1]], [[0,0],[-1,0],[0,0]], [[1,1],[0,1],[0,0]]],
                     [[0,0],[-1,0],[0,0]], [[1,1],[1,1],[1,1]], [[0,0],[-1,0],[0,0]]],
                     [[1,1],[1,1],[0,0]], [[0,0],[-1,0],[0,0]], [[1,1],[0,1],[-1,-1]]
                   ] )
maps = np.swapaxes(weight, 1, 2)
maps = np.swapaxes(maps, 0, 1)

for map in maps:
    map = np.swapaxes(map, 1, 2)
    map = np.swapaxes(map, 0, 1)
    for filter in map:
       print(filter)
```

(3,3,3,2)

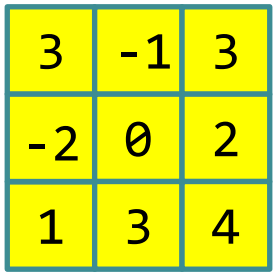
(3,3,2)

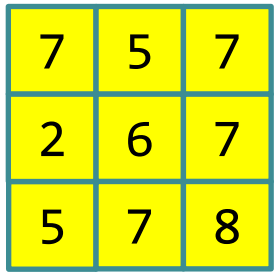
(2,3,3)



```
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=2, kernel_size=3, padding='valid',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)  # ( 1,3,3,2)
feature_maps = np.swapaxes(conv2d, 0, 3)
for feature_map in feature_maps:
    print(feature_map.reshape(3,3))
```

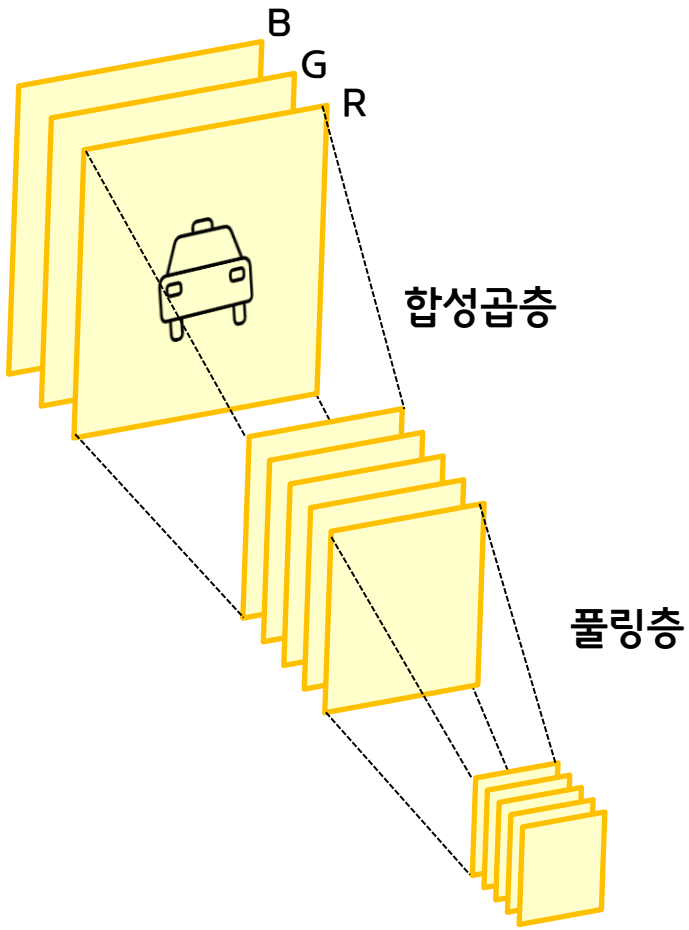
```
[[7. 5. 7.]
 [2. 6. 7.]
 [5. 7. 8.]]
```





풀링 연산

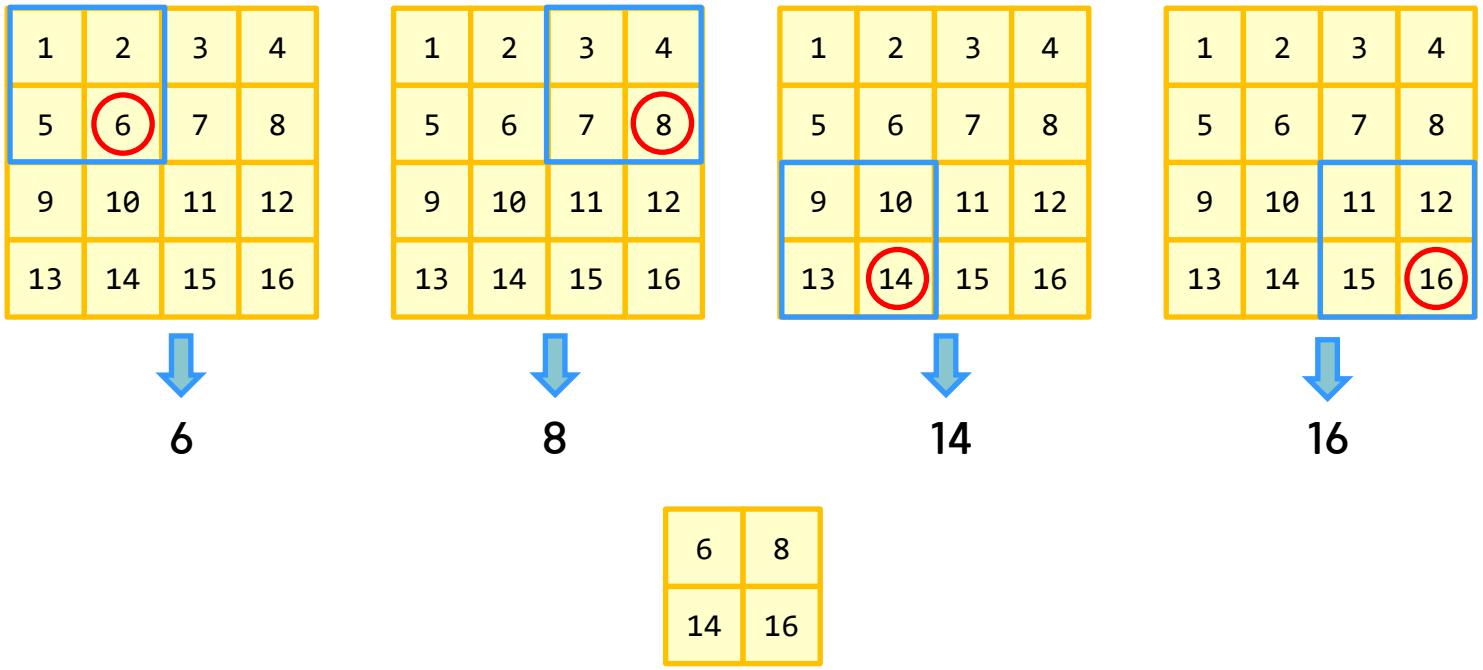
합성곱층과 풀링층을 거치면서 변환되는 과정



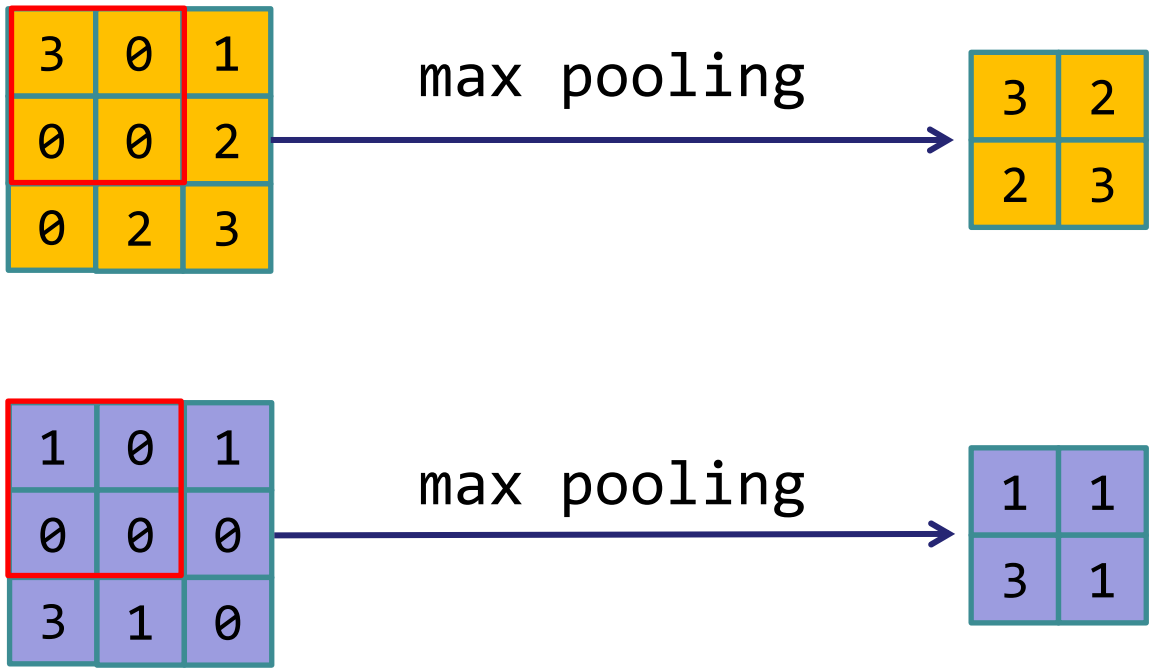
풀링 연산

풀링이란? 특성 맵을 스캔하며 최대값을 고르거나 평균값을 계산하는 것을 말함

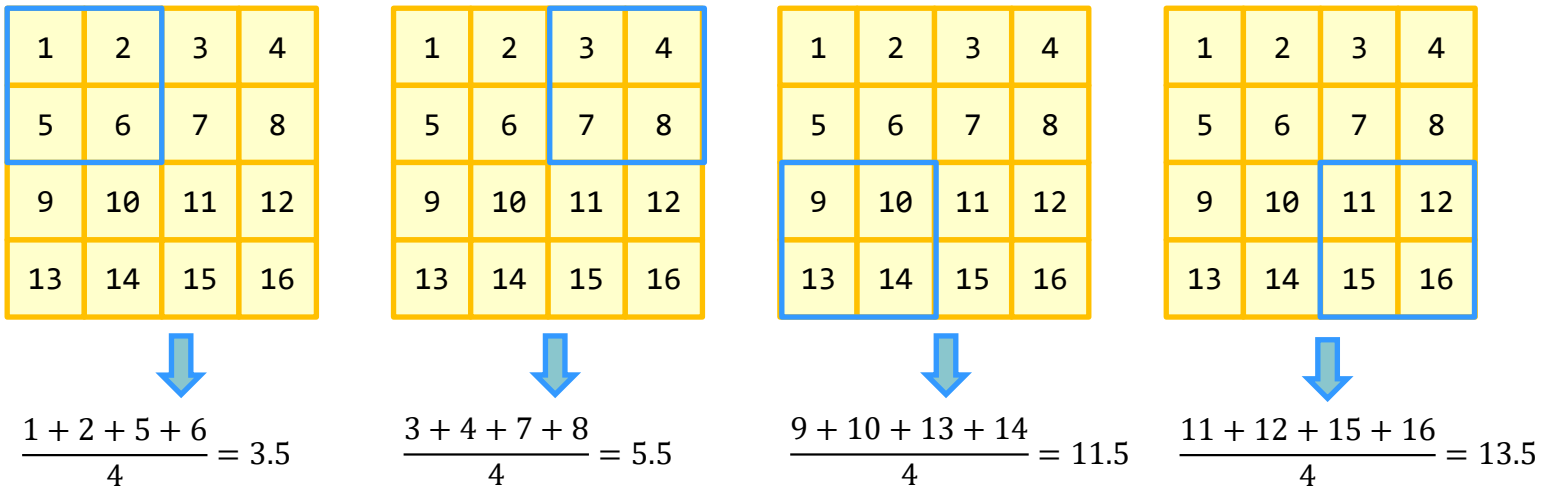
최대 풀링



Pooling(max pooling, 2x2 filter, stride 1)



Pooling(average pooling, 2x2 filter, stride 2)

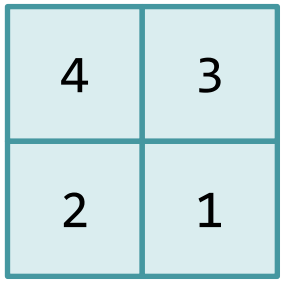


tf.keras.layers.MAXPool2D

- **`pool_size`**: integer or tuple of 2 integers, window size over which to take the maximum. `(2, 2)` will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
- **`strides`**: Integer, tuple of 2 integers, or None. Strides values. Specifies how far the pooling window moves for each pooling step. If None, it will default to `pool_size`.
- **`padding`**: One of `"valid"` or `"same"` (case-insensitive). "valid" adds no zero padding. "same" adds padding such that if the stride is 1, the output shape is the same as input shape.
- **`data_format`**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

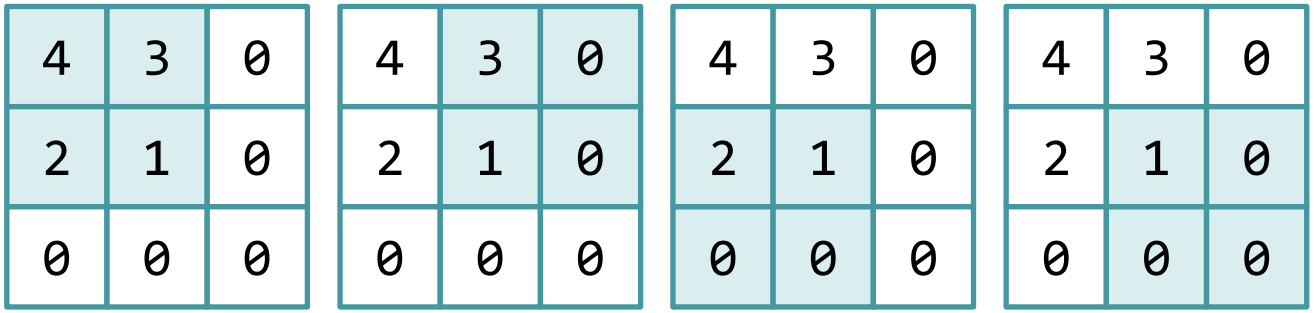
```
image = tf.constant([[[[4],[3]],[[2],[1]]]], dtype=np.float32)
pool = tf.keras.layers.MaxPool2D(pool_size=(2,2), strides=1, padding='valid')(image)
print(pool.shape)
print(pool.numpy())
```

```
(1, 1, 1, 1)
[[[[4.]]]]
```



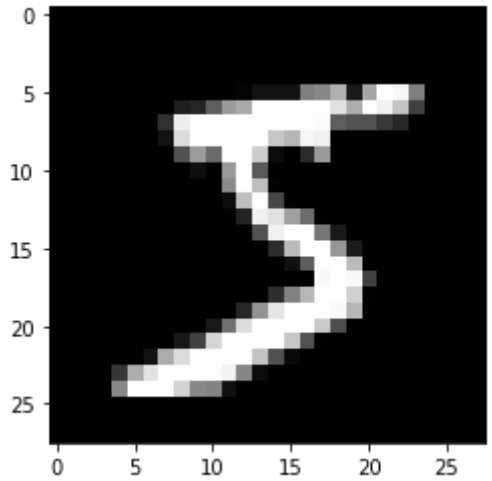
```
image = tf.constant([[[[4],[3]],[[2],[1]]]], dtype=np.float32)
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=1, padding='same')(image)
print(pool.shape)
print(pool.numpy())
```

```
(1, 2, 2, 1)
[[[[4.]
   [3.]]

  [[2.]
   [1.]]]]
```



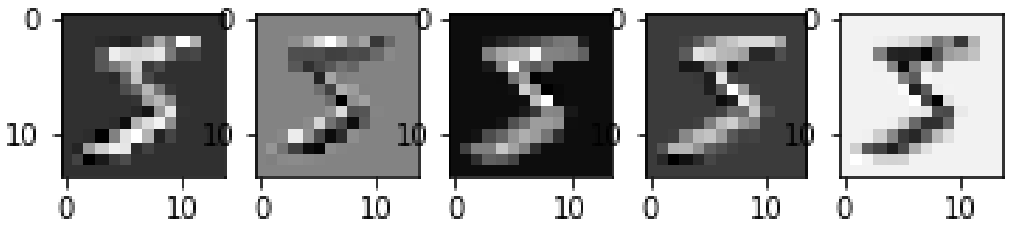
Loading MNIST Data

```
mnist = keras.datasets.mnist
class_names = ['0','1','2','3','4','5','6','7','8','9']

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.astype(np.float32) / 255.
test_images = test_images.astype(np.float32) / 255.

img = train_images[0]
plt.imshow( img, cmap='gray')
plt.show()
```



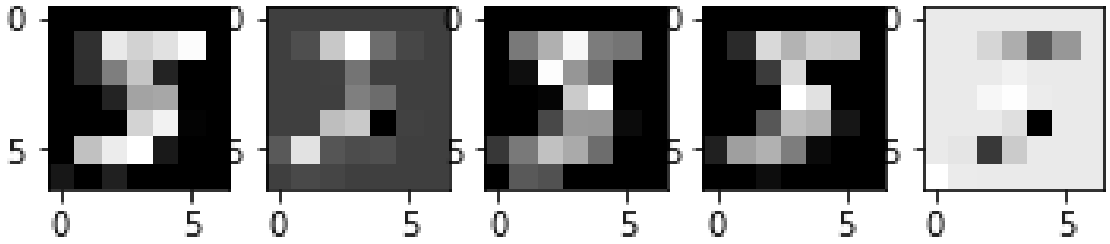
```
img = img.reshape(-1,28,28,1)
img = tf.convert_to_tensor(img)

print("weight.shpe", weight.shape)
weight_init = keras.initializers.RandomNormal(stddev=0.01)
conv2d = keras.layers.Conv2D(filters=5, kernel_size=3, padding='same',
                             strides=(2,2), kernel_initializer=weight_init)(img)
print("conv2d.shape", conv2d.shape)
feature_maps = np.swapaxes(conv2d, 0, 3)
for i, feature_map in enumerate(feature_maps):
    plt.subplot(1,5,i+1), plt.imshow(feature_map.reshape(14,14), cmap='gray')
plt.show()
```

```
weight.shpe (2, 2, 1, 3)
conv2d.shape (1, 14, 14, 5)
```



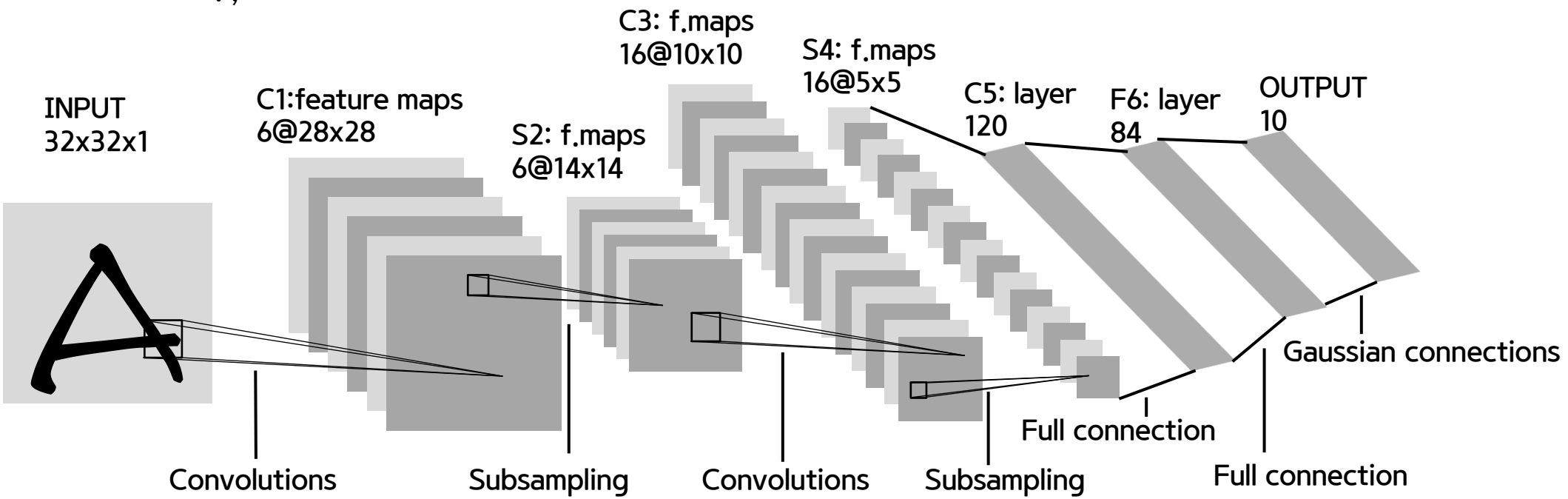
```
pool = keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding='same')(conv2d)
print(pool.shape)
feature_maps = np.swapaxes(pool, 0, 3)
for i, feature_map in enumerate(feature_maps):
    plt.subplot(1,5,i+1), plt.imshow(feature_map.reshape(7,7), cmap='gray')
plt.show()
```

```
(1, 7, 7, 5)
```



LeNet-5

LeCun et al. , 1998



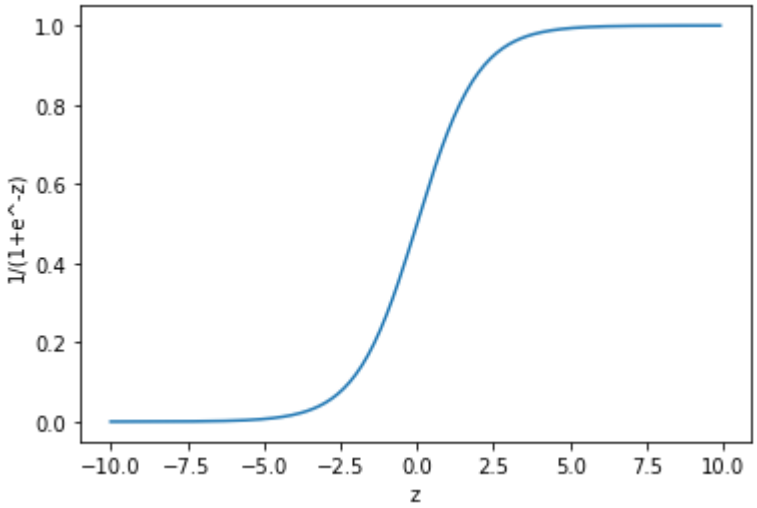
Conv filters 5x5, stride 1
Subsampling 2x2, stride 2

활성화 함수

활성함수는 네트워크에 비선형성(nonlinearity)을 추가하기 위해 사용됨
- 활성함수 없이 layer를 쌓은 네트워크는 1-layer 네트워크와 동일하기 때문에 활성 함수는 비성형함수로 불리기도 한다.
- 멀티레이어 퍼셉트론을 만들 때 활성함수를 사용하지 않으면 쓰나마나이다.

1. 시그모이드 함수 (Sigmoid Function)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



- 결과값이 [0,1] 사이로 제한됨
- 뇌의 뉴런과 유사하여 많이 쓰였음

- 문제점

1) 그레디언트가 죽는 현상이 발생한다 (Gradient vanishing 문제)

gradient 0이 곱해 지니까 그 다음 layer로 전파되지 않는다. 즉, 학습이 되지 않는다.

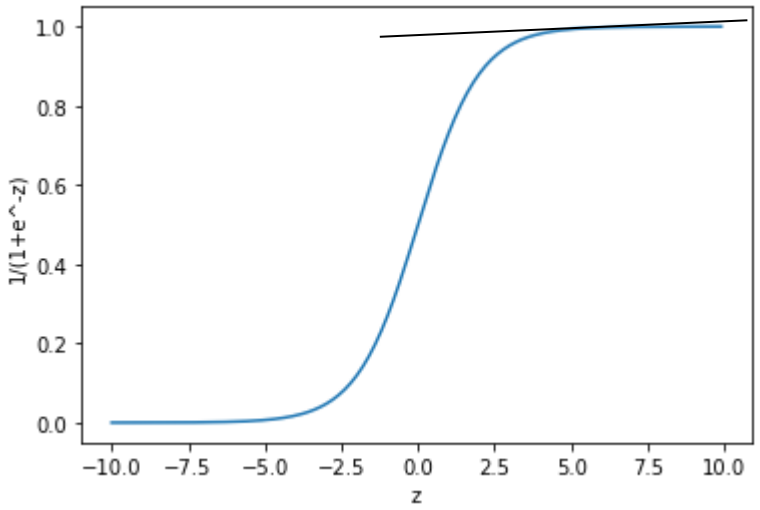
2) 활성함수의 결과 값의 중심이 0이 아닌 0.5이다.

3) 계산이 복잡하다 (지수함수 계산)

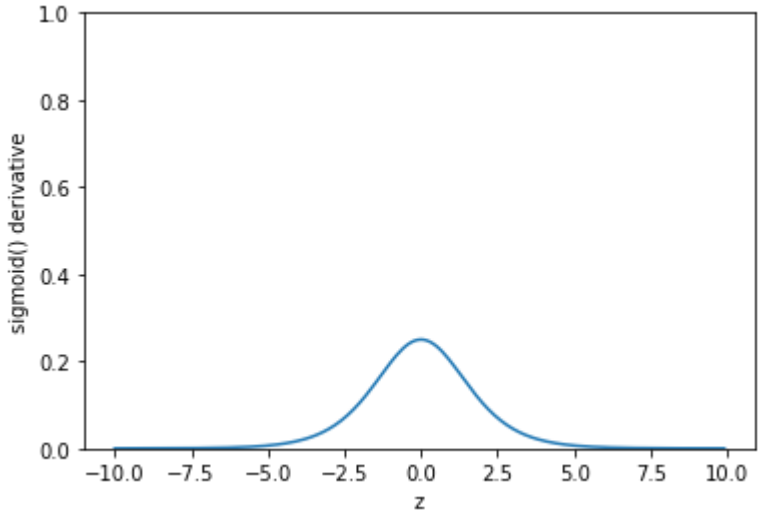
!! Gradient Vanishing

- 시그모이드와 같이 결과값이 포화(saturated)되는 함수는 gradient vanishing 현상을 야기
- 이전 레이어로 전파되는 그라디언트가 0에 가까워 지는 현상
- 레이어를 깊게 쌓으면 파라미터의 업데이트가 제대로 이루어지지 않음
- 양 극단의 미분값이 0에 가깝기 때문에 발생하는 문제

시그모이드 함수



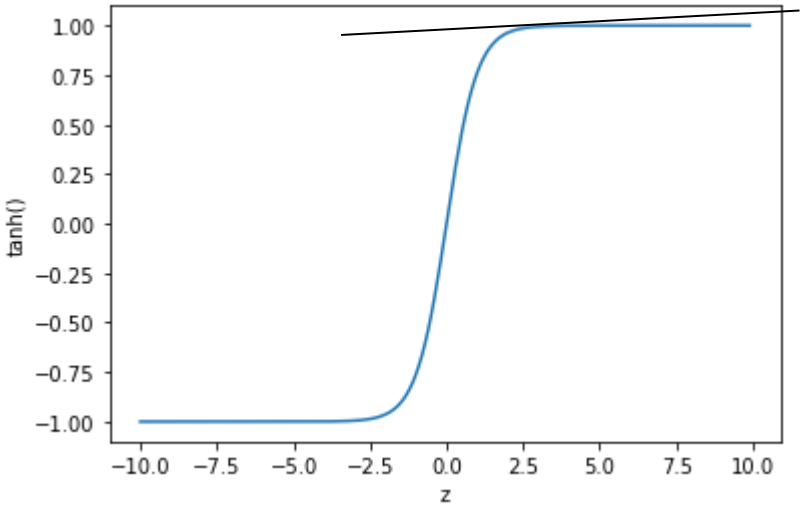
시그모이드 미분값



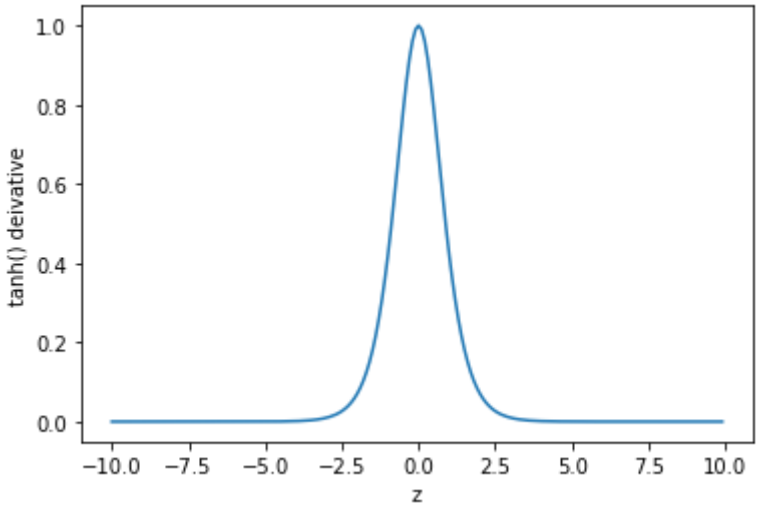
양쪽 꼬리가 0에 수렴하며 최대값이 0.25를 넘지 않는다.

2. 하이퍼 볼릭 탄젠트(tanh)

$$\tanh(x) = 2 * \text{sigmoid}(2 * x) - 1$$



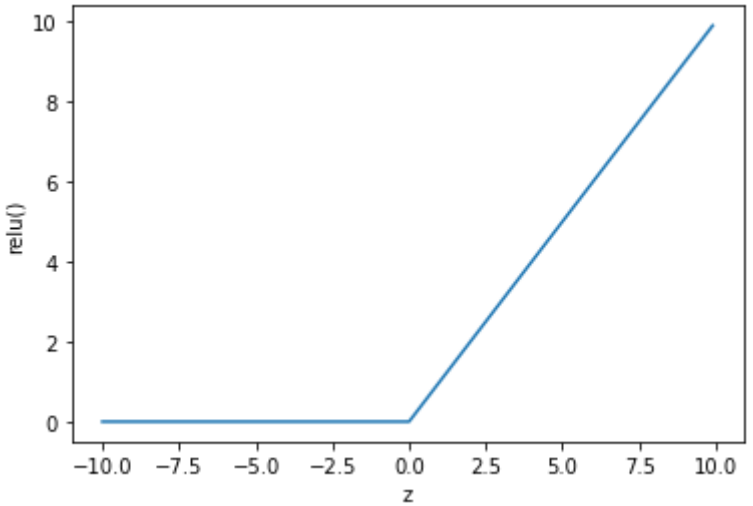
$$\tanh(x)' = (1 - \tanh(x))(1 + \tanh(x))$$



- 결과값이 [-1, 1] 사이로 제한됨. 결과값 중심이 0이다.
- 나머지 특성은 시그모이드와 비슷함. 시그모이드 함수를 이용하여 유도 가능
- 그러나, 여전히 gradient vanishing 문제가 발생

3. 렐루(ReLU, Rectified Linear Unit)

$$f(x) = \max(0, x)$$

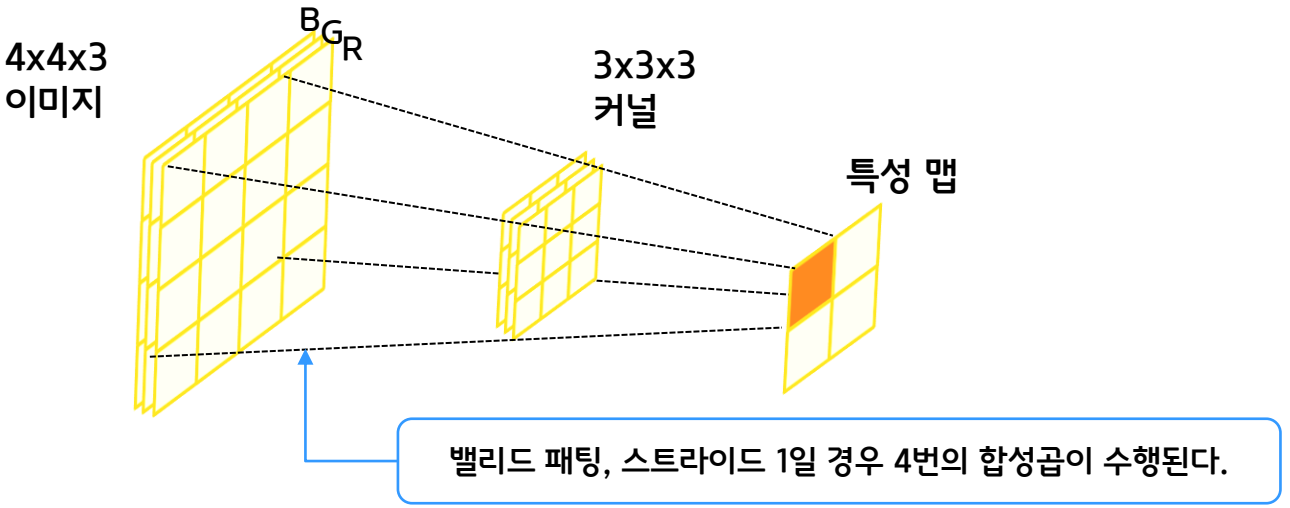


최근 뉴럴 네트워크에서 가장 많이 쓰이는 활성 함수
선형아니야? NO! 0에서 확 꺾이기 때문에 비선형이라고 본다.

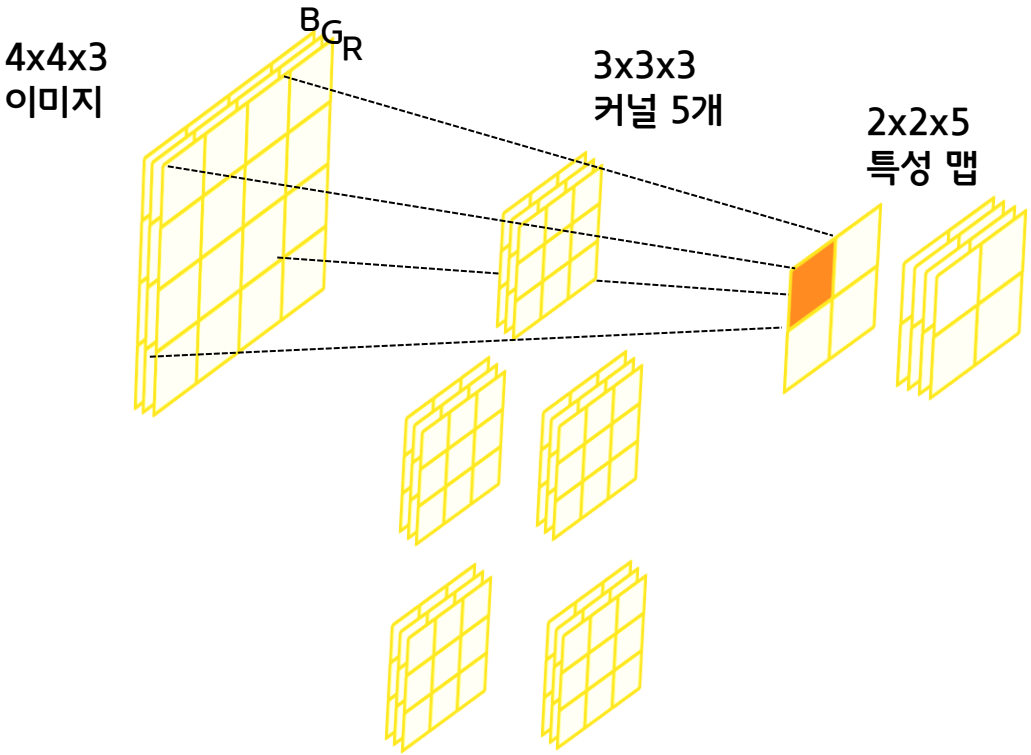
- 장점
(1) 양 극단값이 포화되지 않는다. (양수 지역은 선형적)
(2) 계산이 매우 효율적이다 (최대값 연산 1개)
(3) 수렴속도가 시그모이드류 함수대비 6배 정도 빠르다.

- 단점
(1) 중심값이 0이 아님 (마이너한 문제)
(2) 입력값이 음수인 경우 항상 0을 출력함 (마찬가지로 파라미터 업데이트가 안됨)

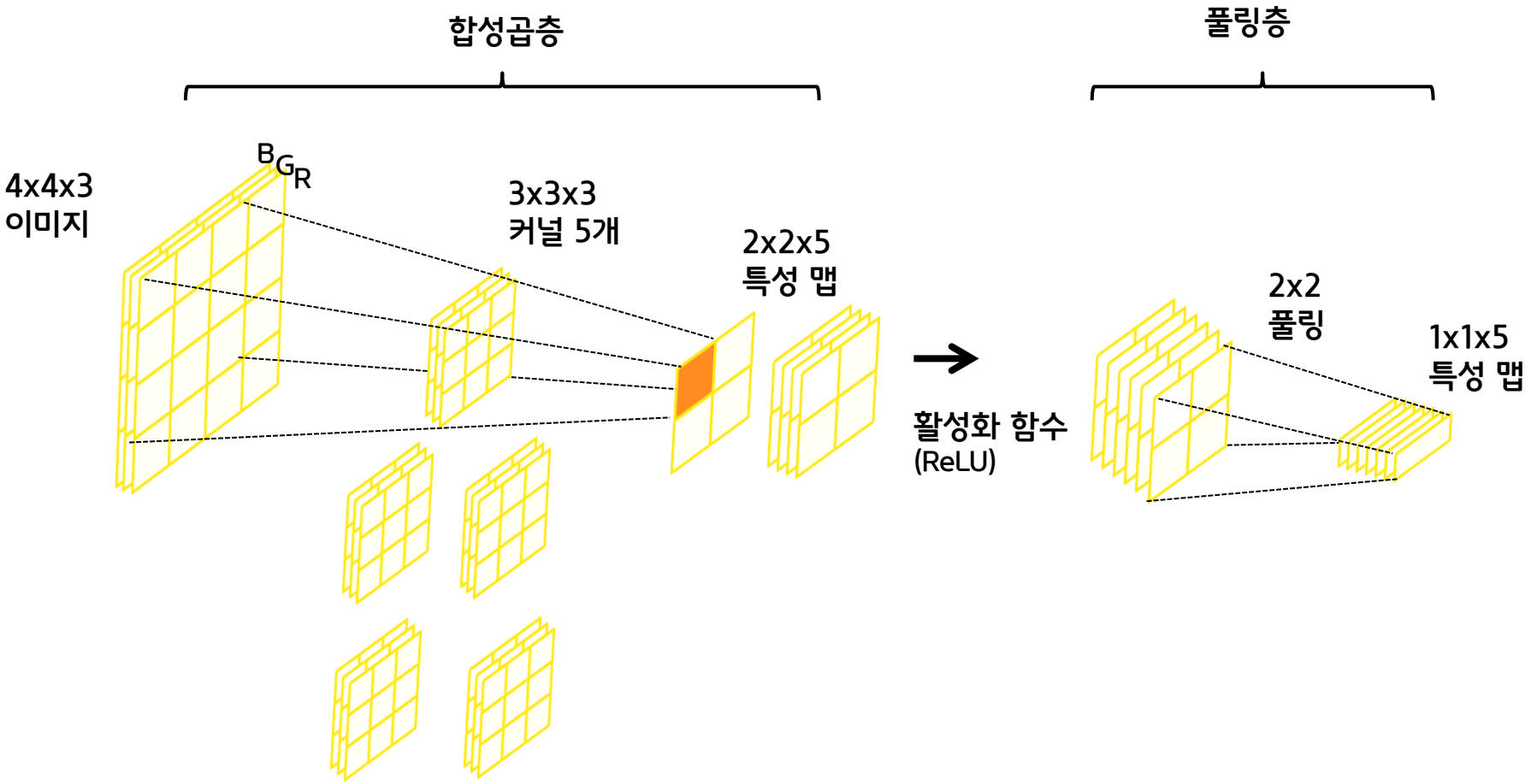
합성곱 층에서 일어나는 일



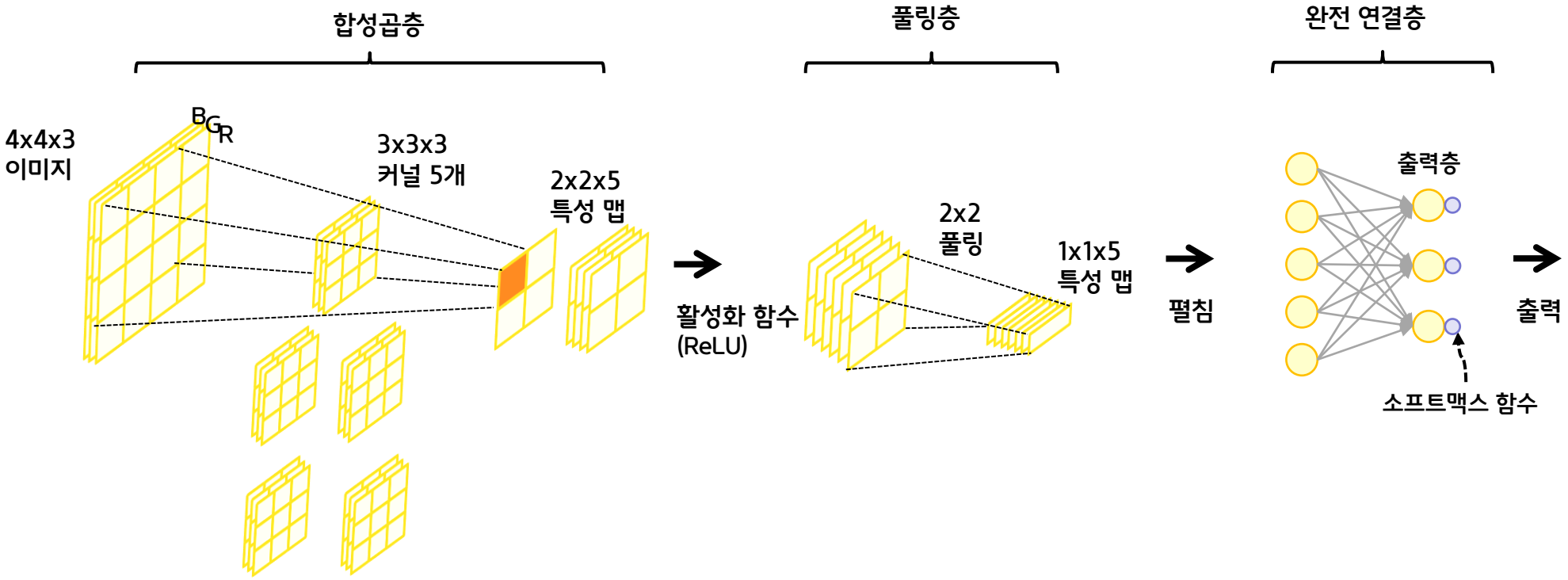
합성곱 층에서 일어나는 일



풀링 층에서 일어나는 일


합성곱층
풀링층
4x4x3
이미지
B
G
R
3x3x3
커널 5개
2x2x5
특성 맵
활성화 함수
(ReLU)
2x2
풀링
1x1x5
특성 맵

특성 맵을 펼쳐 완전 연결 신경망에 주입한다.



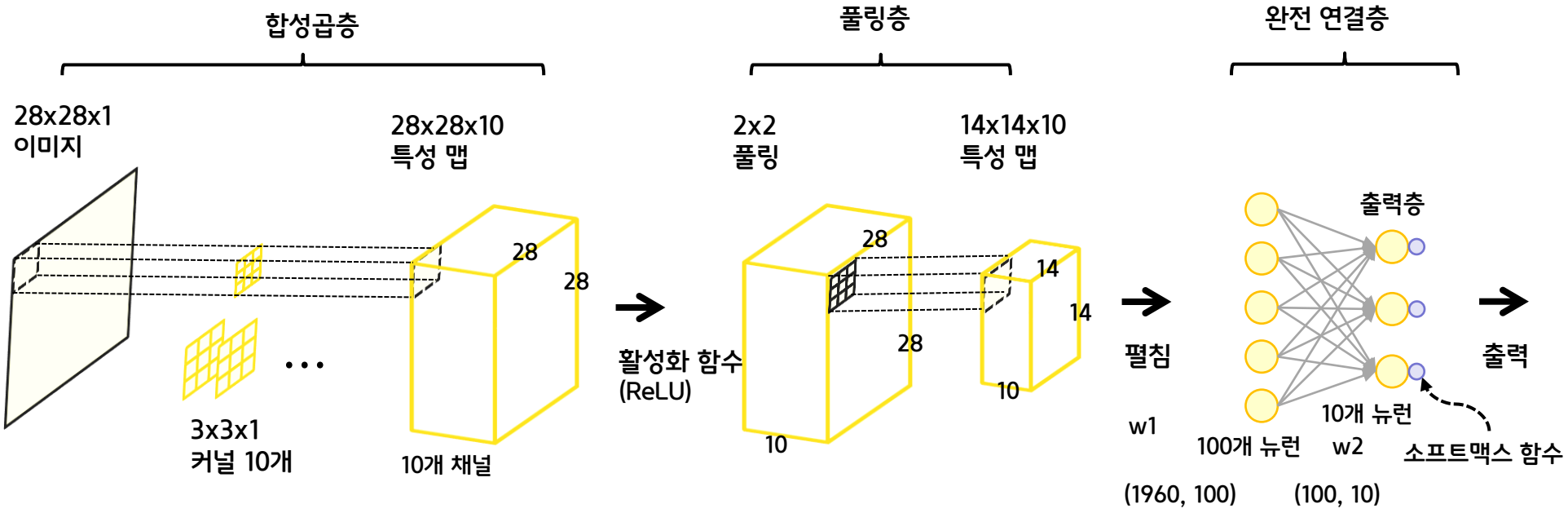
5. 합성곱 신경망 이해

5.1 합성곱 연산

5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 구현

합성곱 신경망의 전체 구조



- 28x28 크기의 흑백 이미지와 3x3 크기의 커널 10개로 합성곱 수행
- 2x2 크기의 최대 풀링을 수행하여 14x14x10로 특성 맵의 크기를 줄인다.
- 특성 맵을 일렬로 펼쳐서 100개의 뉴런을 가진 완전 연결층과 연결 시킨다.
- 10개의 클래스를 구분하기 위한 소프트맥스 함수에 연결한다.

합성곱 신경망의 정방향 구현

합성곱 적용

```
def forpass(self, x):
        # 3x3 합성곱 연산을 수행합니다.
        c_out = tf.nn.conv2d(x, self.conv_w, strides=1, padding='SAME') + self.conv_b
```

- self.conv_w
  self.conv_w는 합성곱에 사용할 가중치이다. 3x3x1 크기의 커널을 10개 사용하므로 가중치의 전체 크기는 3x3x1x10 이다.

- strides, padding
  특성 맵의 가로와 세로 크기를 일정하게 만들기 위하여 strides는 1, padding은 'SAME'으로 지정한다.

렐루 함수 적용

```
def forpass(self, x):
        ...
        r_out = tf.nn.relu(c_out)
```

(1,14,14,10)

합성곱 신경망의 정방향 구현

(1,1960) => (m, n) => XW + b

풀링 적용하고 완전 연결층 수정

self.w1.shape => (1960, 100

```
def forpass(self, x):
      ...
      p_out = tf.nn.max_pool2d(r_out, ksize=2, strides=2, padding='VALID')
      # 첫 번째 배치 차원을 제외하고 출력을 일렬로 펼칩니다.
      f_out = tf.reshape(p_out, [x.shape[0], -1])
      z1 = tf.matmul(f_out, self.w1) + self.b1     # 첫 번째 층의 선형 식을 계산합니다
      a1 = tf.nn.relu(z1)                          # 활성화 함수를 적용합니다
      z2 = tf.matmul(a1, self.w2) + self.b2        # 두 번째 층의 선형 식을 계산합니다.
      return z2
```

- max_pool2d() 함수를 사용하여 2x2 크기의 풀링을 적용 한다.
  이 단계에서 만들어진 특성 맵의 크기는 14x14x10 이다.

- tf.reshape() 함수를 사용해 일렬로 펼친다.
  이때 배치 차원을 제외한 나머지 차원만 펼쳐야 한다.

- np.dot() 함수를 텐서플로의 tf.matmul() 함수로 바꿔서 구현한다.
  이는 conv2d()와 max_pool2d() 등이 Tensor 객체를 반환하기 때문임

- 완전 연결층의 활성화 함수도 시그모이드 대신 렐루 함수를 사용한다.

합성곱 신경망의 역방향 계산 구현

자동 미분의 사용 방법

```
x = tf.Variable(np.array([1.0, 2.0, 3.0]))
with tf.GradientTape() as tape:
    y = x ** 3 + 2 * x + 5
# 그래디언트를 계산합니다.
print(tape.gradient(y, x))
```

```
tf.Tensor([ 5. 14. 29.], shape=(3,), dtype=float64)
```

- 텐서플로와 같은 딥러닝 패키지들은 사용자가 작성한 연산을 계산 그래프로 만들어 자동 미분 기능을 구현한다.
- 자동 미분기능을 사용하면 임의의 파이썬 코드나 함수에 대한 미분값을 계산할 수 있다.
- 텐서플로의 자동 미분 기능을 사용하려면 with블럭으로 tf.GradientTape() 객체가 감시할 코드를 감싸야 한다.
- tape객체는 with블럭 안에서 일어나는 모든 연산을 기록하고 텐서플로 변수인 tf.Variable객체를 자동으로 추적한다.
- 그레이디언트를 계산하려면 미분 대상 객체와 변수를 tape객체의 gradient() 함수에 전달해야 한다.

$x^3 + 2x + 5$ 를 미분하면 $3x^2 + 2$ 가 되므로

1.0, 2.0, 3.0을 미분 방정식에 대입하면 5.0, 14.0, 29.0 을 얻는다.

합성곱 신경망의 역방향 계산 구현

1. 역방향 계산 구현

```
def training(self, x, y):
        m = len(x)                      # 샘플 개수를 저장합니다.
        with tf.GradientTape() as tape:
            z = self.forpass(x)         # 정방향 계산을 수행합니다.
            # 손실을 계산합니다.
            loss = tf.nn.softmax_cross_entropy_with_logits(y, z)
            loss = tf.reduce_mean(loss)
```

- 자동 미분 기능을 사용하면 ConvolutionNetwork의 backprop() 함수를 구현할 필요가 없다.

- training() 함수에서 forpass() 함수를 호출하여 정방향 계산을 수행한 다음

- tf.nn.softmax_cross_entropy_with_logits() 함수를 호출하여 정방향 계산의 결과(z)와 타깃(y)을 기반으로 손실값을 계산한다.

- 이렇게 하면 크로스 엔트로피 손실과 그레이디언트 계산을 올바르게 처리해 주므로 편하다.

- softmax_cross_entropy_with_logits() 함수는 배치의 각 샘플에 대한 손실을 반환하므로 reduce_mean() 함수로 평균을 계산한다.

합성곱 신경망의 역방향 계산 구현

2. 그레이디언트 계산

```
def training(self, x, y):
        ...
        weights_list = [self.conv_w, self.conv_b,
                        self.w1, self.b1, self.w2, self.b2]
        # 가중치에 대한 그래디언트를 계산합니다.
        grads = tape.gradient(loss, weights_list)
        # 가중치를 업데이트합니다.
        self.optimizer.apply_gradients(zip(grads, weights_list))
```

- tape.gradient() 를 이용하면 그레이디언트를 자동으로 계산할 수 있다.
- 합성곱층의 가중치와 절편인 con_w와 con_b를 포함하여 그레이디언트가 필요한 weights_list로 나열한다.
- 텐서플로의 옵티마이저를 사용하면 간단하게 알고리즘을 바꾸어 테스트할 수 있다.
- self.optimizer.apply_gradients() 함수에는 그레이디언트와 가중치를 튜플로 묶은 리스트를 전달해야 한다.

옵티마이저 객체를 만들어 가중치 초기화

1. fit() 함수 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
        self.init_weights(x.shape, y.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
        self.optimizer = tf.optimizers.SGD(learning_rate=self.lr)
        # epochs만큼 반복합니다.
        for i in range(epochs):
            print('에포크', i, end=' ')
            # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
            batch_losses = []
            for x_batch, y_batch in self.gen_batch(x, y):
                print('.', end='')
                self.training(x_batch, y_batch)
                # 배치 손실을 기록합니다.
                batch_losses.append(self.get_loss(x_batch, y_batch))
            print()
            # 배치 손실 평균내어 훈련 손실 값으로 저장합니다.
            self.losses.append(np.mean(batch_losses))
            # 검증 세트에 대한 손실을 계산합니다.
            self.val_losses.append(self.get_loss(x_val, y_val))
```

- 텐서플로는 tf.optimizers 모듈 아래에 여러 종류의 경사 하강법을 구현해 놓았다.

- SGD옵티마이저(tf.optimizers.SGD)객체는 기본 경사 하강법이다.

옵티마이저 객체를 만들어 가중치 초기화

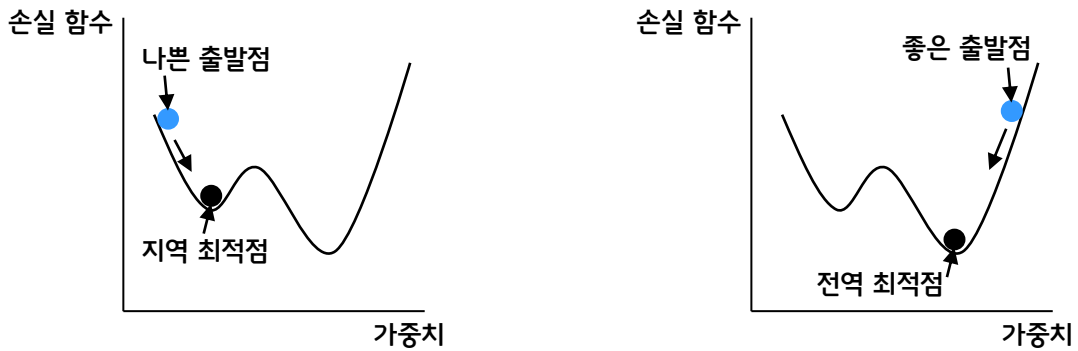
2. init_weights() 함수 수정

```
def init_weights(self, input_shape, n_classes):
        g = tf.initializers.glorot_uniform()
        self.conv_w = tf.Variable(g((3, 3, 1, self.n_kernels)))
        self.conv_b = tf.Variable(np.zeros(self.n_kernels), dtype=float)
        n_features = 14 * 14 * self.n_kernels
        self.w1 = tf.Variable(g((n_features, self.units)))          # (특성 개수, 은닉층의 크기)
        self.b1 = tf.Variable(np.zeros(self.units), dtype=float)    # 은닉층의 크기
        self.w2 = tf.Variable(g((self.units, n_classes)))           # (은닉층의 크기, 클래스 개수)
        self.b2 = tf.Variable(np.zeros(n_classes), dtype=float)     # 클래스 개수
```

- 가중치를 glorot_uniform() 함수로 초기화 한다.
- 가중치를 tf.Variable() 함수로 만들어야 한다.
- np.zeros는 64bit로 초기화 되므로 dtype=float으로 32bit 바꿔준다.

옵티마이저 객체를 만들어 가중치 초기화

glorot_uniform()에 대하여



- 경사 하강법은 출발점으로부터 기울기가 0이 되는 최적점을 찾아 간다.
- 출발점이 적절하지 않으면 엉뚱한 곳을 최적점으로 판단할 수 있다. 이런 지점을 지역 최저점이라 한다.
- 가중치를 적절하게 초기화 했다면 올바른 최적점을 찾게 된다. 이런 지점을 전역 최저점이라 한다.

옵티마이저 객체를 만들어 가중치 초기화

glorot_uniform()에 대하여

- 글로럿 초기화 방식은 세이비어 글로럿(Xavier Glorot)이 제안하여 널리 사용되고 있다.
- 텐서 플로의 glorot_uniform() 함수는

$$-\sqrt{\frac{6}{\text{입력뉴런 수} + \text{출력 뉴런수}}} \sim +\sqrt{\frac{6}{\text{입력뉴런 수} + \text{출력 뉴런수}}}$$

사이에서 균등하게 난수를 발생시켜 가중치를 초기화 한다.

- 합성곱 영역의 너비와 높이는 3x3이고 흑백 이미지의 입력 채널은 하나이므로 커널의 크기는 3x3x1이 된다.
- 합성곱 커널을 n_kernels만큼 만들기 위해 3x3x1xn_kernels 크기의 4차원 배열로 초기화 한다.
- 합성곱과 풀링층을 거치면 입력 이미지의 높이와 너비가 28에서 14로 줄어 든다.
-이에 필요한 가중치 w1의 크기는 14x14xn_kernels가 된다.

합성곱 신경망 훈련

1. 데이터 세트 불러오기

```
(x_train_all, y_train_all), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

2. 훈련 세트와 검증 세트로 나누기

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train_all, y_train_all, stratify=y_train_all,
                                                  test_size=0.2, random_state=42)
```

3. 타깃을 원-핫 인코딩으로 변환하기

```
y_train_encoded = tf.keras.utils.to_categorical(y_train)
y_val_encoded = tf.keras.utils.to_categorical(y_val)
```

4. 입력 데이터 준비하기

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_val = x_val.reshape(-1, 28, 28, 1)
```

```
x_train.shape
```

(48000, 28, 28, 1)

합성곱 신경망 훈련

5. 입력 데이터 표준화 전처리하기

```
x_train = x_train / 255
x_val = x_val / 255
```

6. 모델 훈련하기

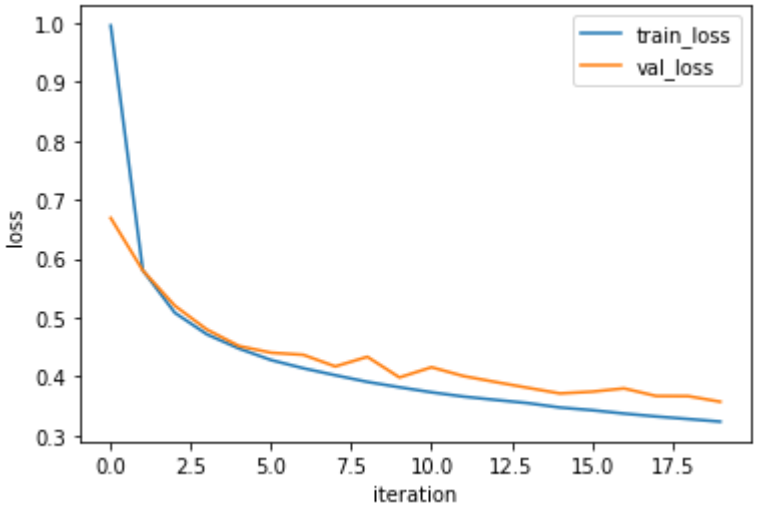
```
cn = ConvolutionNetwork(n_kernels=10, units=100, batch_size=128, learning_rate=0.01)
cn.fit(x_train, y_train_encoded,
       x_val=x_val, y_val=y_val_encoded, epochs=20)
```

```
에포크 0 ..........
에포크 1 ..........
.
.
.
에포크 19 ..........
```

합성곱 신경망 훈련

7. 훈련, 검증 손실 그래프 그리고 검증 세트의 정확도 확인

```
import matplotlib.pyplot as plt
plt.plot(cn.losses)
plt.plot(cn.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```




```
cn.score(x_val, y_val_encoded)
```

0.87725

5. 합성곱 신경망 이해

5.1 합성곱 연산

5.2 합성곱 신경망 구현

5.3 케라스로 합성곱 신경망 구현

케라스로 합성곱 신경망 만들기

1. 필요한 클래스들을 임포트하기

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

2. 합성곱층 쌓기

```
conv1 = tf.keras.Sequential()
conv1.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
```

- Conv2D클래스의 첫 번째 매개변수는 합성곱 커널의 개수이다.
- 두번째 매개변수는 합성곱 커널의 크기로 높이와 너비를 튜플로 전달한다.
- activation 매개변수에 렐루 활성화 함수를 지정한다.
- padding은 same을 지정한다. 이때 대소문자를 구분하지 않는다.
- Sequential 클래스에 층을 처음 추가할 때는 배치 차원을 제외한 입력의 크기를 지정한다.

3. 풀링층 쌓기

```
conv1.add(MaxPooling2D((2, 2)))
```

- MaxPooling2D 클래스의 첫 번째 매개변수는 풀링의 높이와 너비를 나타내는 튜플이며, 스트라이드는 strides에 지정할 수 있음
- 패딩은 padding에 지정하며 기본값은 'valid'이다.

4. 완전 연결층에 주입할 수 있도록 특성 맵 펼치기

```
conv1.add(Flatten())
```

- 풀링 다음에는 완전 연결층에 연결하기 위해 배치 차원을 제외하고 일렬로 펼쳐야 한다. 이 일은 Flatten 클래스가 수행함

케라스로 합성곱 신경망 만들기

5. 완전 연결층 쌓기

```
conv1.add(Dense(100, activation='relu'))
conv1.add(Dense(10, activation='softmax'))
```

- 첫 번째 완전 연결층에는 100개의 뉴런을 사용하고 렐루 활성화 함수를 적용한다.
- 마지막 출력층에는 10개의 클래스에 대응하는 10개의 뉴런을 사용하고 소프트맥스 활성화 함수를 적용한다.

6. 모델 구조 살펴보기

```
conv1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 10)        100       ( 3x3x1x10+10 )
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 10)        0
_________________________________________________________________
flatten (Flatten)            (None, 1960)              0         ( 14x14x10 )
_________________________________________________________________
dense (Dense)                (None, 100)               196100    ( 1960x100+100 )
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010      ( 100x10+10 )
=================================================================
Total params: 197,210
Trainable params: 197,210
Non-trainable params: 0
```

합성곱 신경망 모델 훈련하기

1. 모델 컴파일

```
conv1.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- 다중 분류를 위한 크로스 엔트로피 손실 함수를 사용한다.
- 정확도를 관찰하기 위해 metrics 매개변수에 'accuracy'를 리스트로 전달한다.
- 아담(adam) 옵티마이저를 사용한다.
  아담은 Adaptive Moment Estimation을 줄여 만든 이름이다.
  아담은 손실 함수의 값이 최적값에 가까워질수록 학습률을 낮춰 손실 함수의 값이 안정적으로 수렴될 수 있게 해준다.

2. 모델 훈련

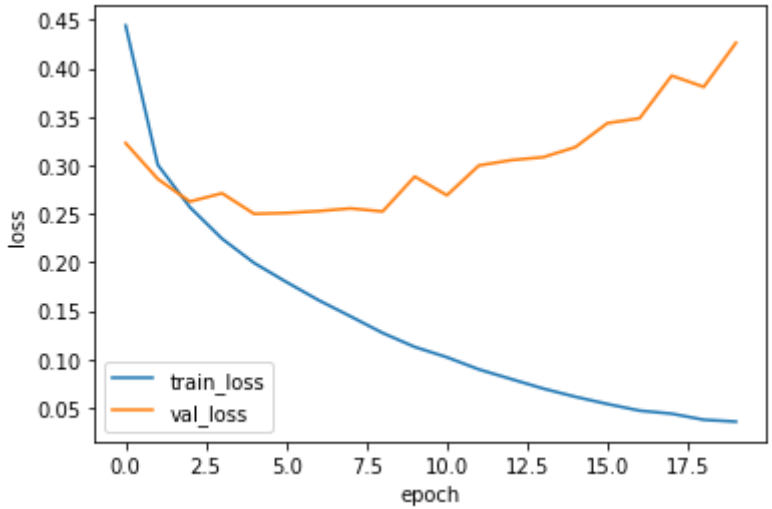
```
history = conv1.fit(x_train, y_train_encoded, epochs=20,
                    validation_data=(x_val, y_val_encoded))
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 8s 171us/sample - loss: 0.4442 - accuracy: 0.8434 - val_loss: 0.3229 - val_accuracy: 0.8862
Epoch 2/20
48000/48000 [==============================] - 8s 166us/sample - loss: 0.3005 - accuracy: 0.8923 - val_loss: 0.2860 - val_accuracy: 0.8968
Epoch 3/20
48000/48000 [==============================] - 8s 160us/sample - loss: 0.2568 - accuracy: 0.9062 - val_loss: 0.2626 - val_accuracy: 0.9043
...
Epoch 20/20
48000/48000 [==============================] - 8s 172us/sample - loss: 0.0356 - accuracy: 0.9875 - val_loss: 0.4264 - val_accuracy: 0.9135
```

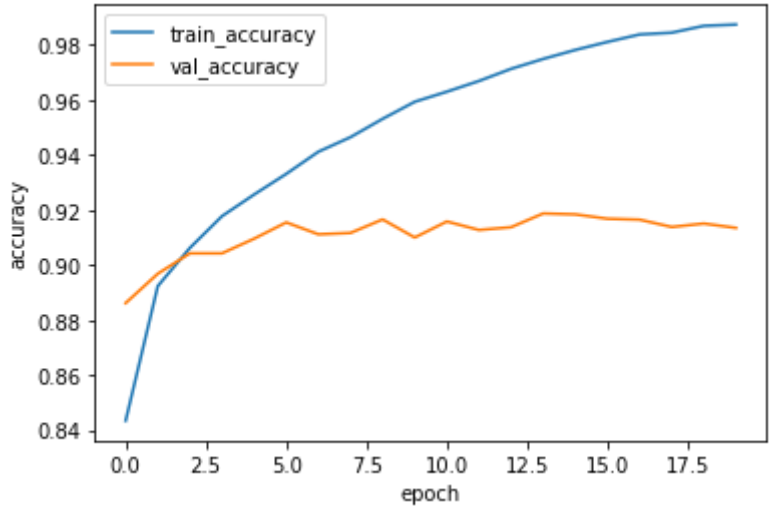
합성곱 신경망 모델 훈련하기

3. 손실 그래프와 정확도 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

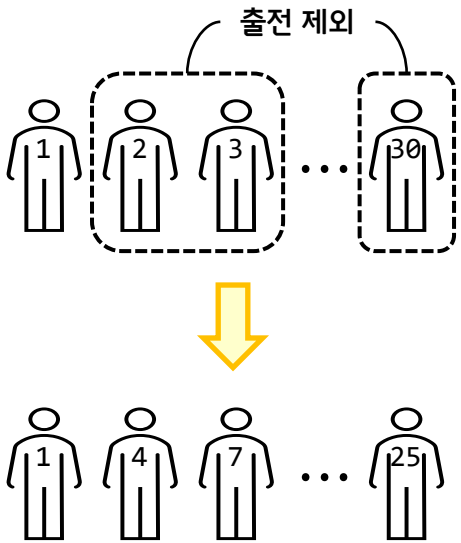


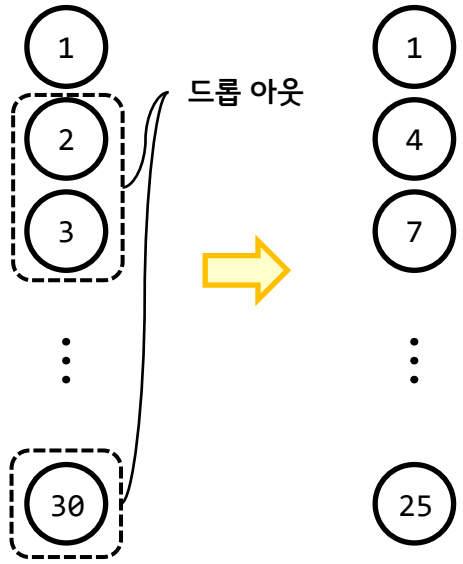

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



드롭아웃

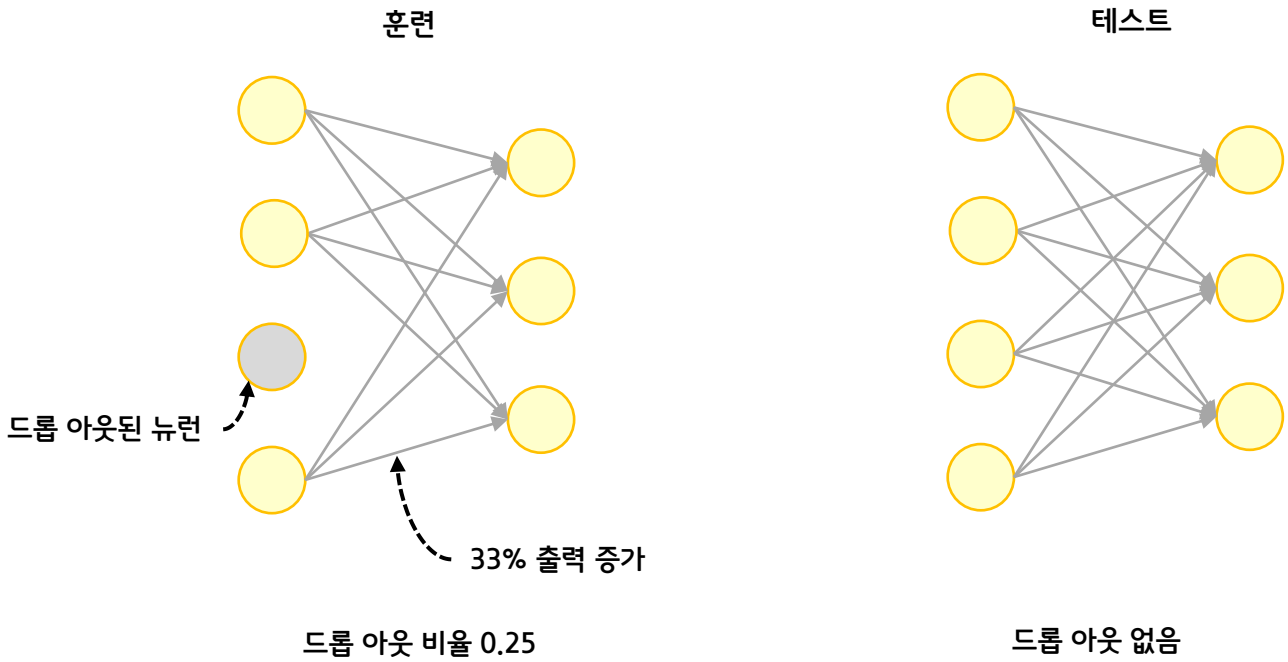
드롭아웃에 대하여





드롭아웃

드롭아웃에 대하여



드롭아웃 적용해 합성곱 신경망 구현

1. 케라스로 만든 합성곱 신경망에 드롭아웃 적용하기

```
from tensorflow.keras.layers import Dropout

conv2 = tf.keras.Sequential()
conv2.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
conv2.add(MaxPooling2D((2, 2)))
conv2.add(Flatten())
conv2.add(Dropout(0.5))
conv2.add(Dense(100, activation='relu'))
conv2.add(Dense(10, activation='softmax'))
```

2. 드롭아웃층 확인하기

```
conv2.summary()
```

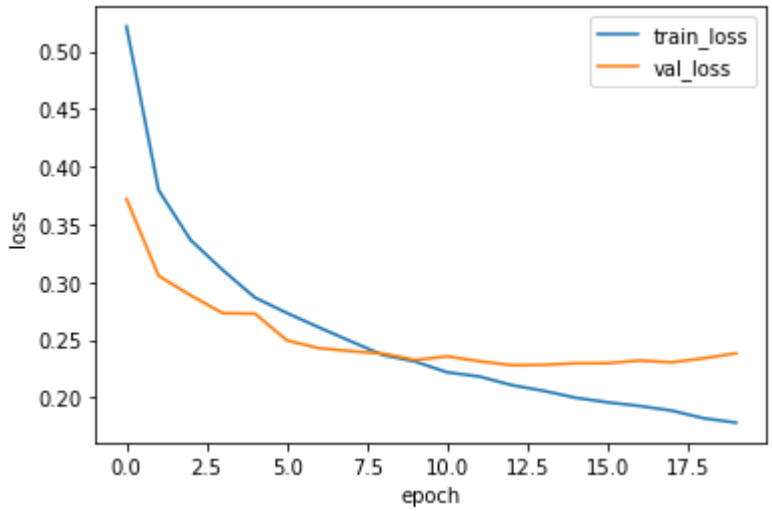
3. 훈련하기

```
conv2.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
history = conv2.fit(x_train, y_train_encoded, epochs=20,
                    validation_data=(x_val, y_val_encoded))
```

드롭아웃 적용해 합성곱 신경망 구현

4. 손실 그래프와 정확도 그래프 그리기

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```